# Advanced Data Structures and Algorithm Analysis

丁尧相
浙江大学

Spring & Summer 2024
Lecture 11
2024-5-6

# Outline: Approximation Algorithms

- Bin packing
- 0-1 Knapsack
- K-center selection
- Take-home messages

# Outline: Approximation Algorithms

- Bin packing
- 0-1 Knapsack
- K-center selection
- Take-home messages

# Computational Complexity

- decision problem $f: \{0,1\}^* \rightarrow \{0,1\}$
- formal language $L \subseteq \{0,1\}^*$ $\qquad L = \{x: f(x)=1\}$
- poly-time Turing machine (Algorithm) $M$:

  length of input

  $\forall$ input $x \in \{0,1\}^*$, $M(x)$ terminates in time < $\mathrm{poly}(|x|)$
- **P**, **NP**: classes of formal languages (decision problems)
  - $L \in \mathbf{P}$: $\exists$ poly-time TM M *decides* $L$
    - $x \in L \Rightarrow M(x)$ accepts;
    - $x \notin L \Rightarrow M(x)$ rejects
  - $L \in \mathbf{NP}$: $\exists$ poly-time TM M *verifies* $L$
    - $x \in L \Rightarrow \exists$ *certificate* $y \in \{0,1\}^*$, $M(x,y)$ accepts;
    - $x \notin L \Rightarrow \forall\ y \in \{0,1\}^*$, $M(x,y)$ rejects;

    *nondeterministic* poly-time TM accepts $L$

# Computational Complexity

- decision problem $f: \{0,1\}^* \to \{0,1\}$
- formal language $L \subseteq \{0,1\}^*$ $\qquad L = \{x: f(x)=1\}$
- poly-time Turing machine (Algorithm) $M$:

  $\forall$ input $x \in \{0,1\}^*$, $M(x)$ terminates in time < $\mathrm{poly}(|x|)$

  length of input

- **P**, **NP**: classes of formal languages (decision problems)
  - $L \in \mathbf{P}$: $\exists$ poly-time TM M *decides* $L$
    - $x \in L \Rightarrow M(x)$ accepts; • $x \notin L \Rightarrow M(x)$ rejects
  - $L \in \mathbf{NP}$: $\exists$ poly-time TM M *verifies* $L$

  $L \in \mathrm{co}\mathbf{NP}$: $\overline{L} \in \mathbf{NP}$ "no"-instances are easy to verify

$$\mathbf{P} \subseteq \mathbf{NP} \cap \mathrm{co}\mathbf{NP}$$

# Computational Complexity

- decision problem $f$: $\{0,1\}^* \to \{0,1\}$
- formal language $L \subseteq \{0,1\}^*$ $\qquad L = \{x: f(x)=1\}$
- poly-time (Turing) *reduction* from $L$ to $L'$:

a poly-time TM $M$ that decides $L$
given accesses to an *oracle* that decides $L'$

$L'$ is poly-time decidable $\Rightarrow$ $L$ is poly-time decidable

$L$ is hard $\Rightarrow$ $L'$ is hard

" $L'$ is at least as hard as $L$ "

- a problem is **NP**-hard if every $L \in \mathbf{NP}$ is poly-time reducible to it
- $L$ is **NP**-complete if $L \in \mathbf{NP}$ and $L$ is **NP**-hard

# Optimization

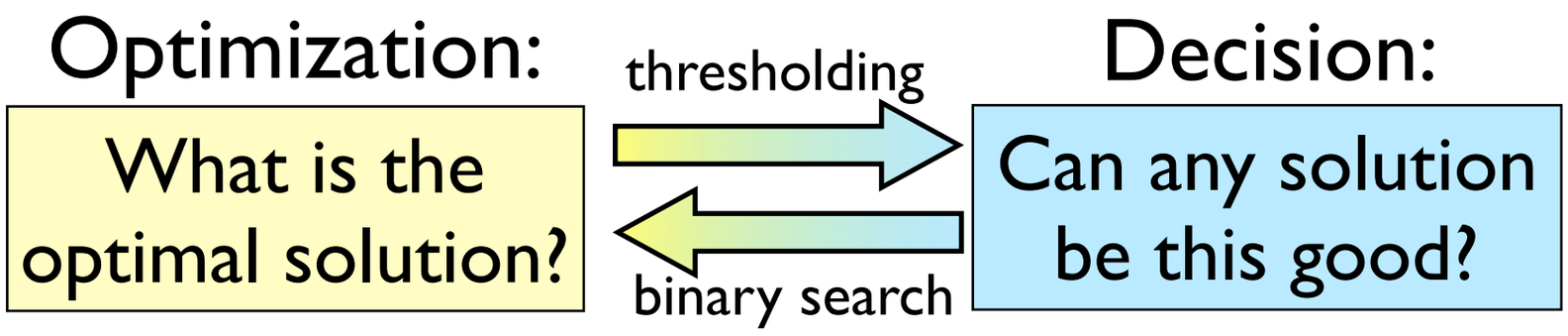
Optimization problem $\Pi$: minimization/maximization

- a set $D$ of valid instances (inputs);
- each instance $I \in D$ defines a set of feasible solutions $S(I)$;
- an objective function $obj$ that assigns each instance $I \in D$ and solution $s \in S(I)$ a value.

**NP**-optimization problem $\Pi$:

- feasibility of a solution is poly-time checkable;
- objective function is poly-time computable.

optimal solution is certificate



# Coping with NP-completeness

Q. Suppose I need to solve an **NP**-hard optimization problem.

What should I do?

# Coping with NP-completeness

Q. Suppose I need to solve an **NP**-hard optimization problem.
What should I do?

A. Sacrifice one of three desired features.

i. Runs in polynomial time.

ii. Solves arbitrary instances of the problem.

iii. Finds optimal solution to problem.

# Coping with NP-completeness

Q. Suppose I need to solve an **NP**-hard optimization problem.
What should I do?

A. Sacrifice one of three desired features.

i. Runs in polynomial time.

ii. Solves arbitrary instances of the problem.

iii. Finds optimal solution to problem.

$\rho$-approximation algorithm.

- Runs in polynomial time.
- Solves arbitrary instances of the problem
- Finds solution that is within ratio $\rho$ of optimum.

Challenge. Need to prove a solution's value is close to optimum, without even knowing what is optimum value.

# Approximate Bin Packing

# Approximate Bin Packing

**Given $N$ items of sizes $S_1, S_2, \ldots, S_N$, such that $0 < S_i \leq 1$ for all $1 \leq i \leq N$. Pack these items in the fewest number of bins, each of which has unit capacity.**

# Approximate Bin Packing

Given $N$ items of sizes $S_1, S_2, \ldots, S_N$, such that $0 < S_i \le 1$ for all $1 \le i \le N$. Pack these items in the fewest number of bins, each of which has unit capacity.

〖Example〗 $N = 7$; $S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8$

# Approximate Bin Packing

Given $N$ items of sizes $S_1, S_2, \ldots, S_N$, such that $0 < S_i \le 1$ for all $1 \le i \le N$. Pack these items in the fewest number of bins, each of which has unit capacity.

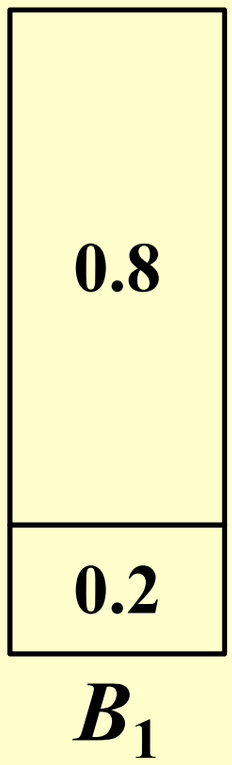
〖Example〗 $N = 7$; $S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8$

## An Optimal Packing

# Approximate Bin Packing

**Given $N$ items of sizes $S_1, S_2, \ldots, S_N$, such that $0 < S_i \le 1$ for all $1 \le i \le N$. Pack these items in the fewest number of bins, each of which has unit capacity.**

**〖Example〗** $N = 7$; $S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8$

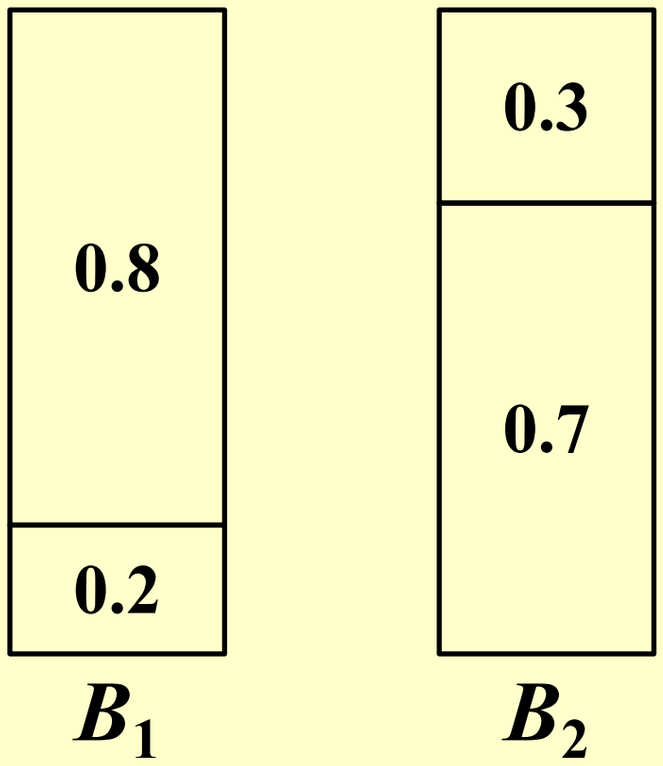



**An Optimal Packing**

# Approximate Bin Packing

**Given $N$ items of sizes $S_1, S_2, \ldots, S_N$, such that $0 < S_i \le 1$ for all $1 \le i \le N$. Pack these items in the fewest number of bins, each of which has unit capacity.**

**〖Example〗** $N = 7$; $S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8$

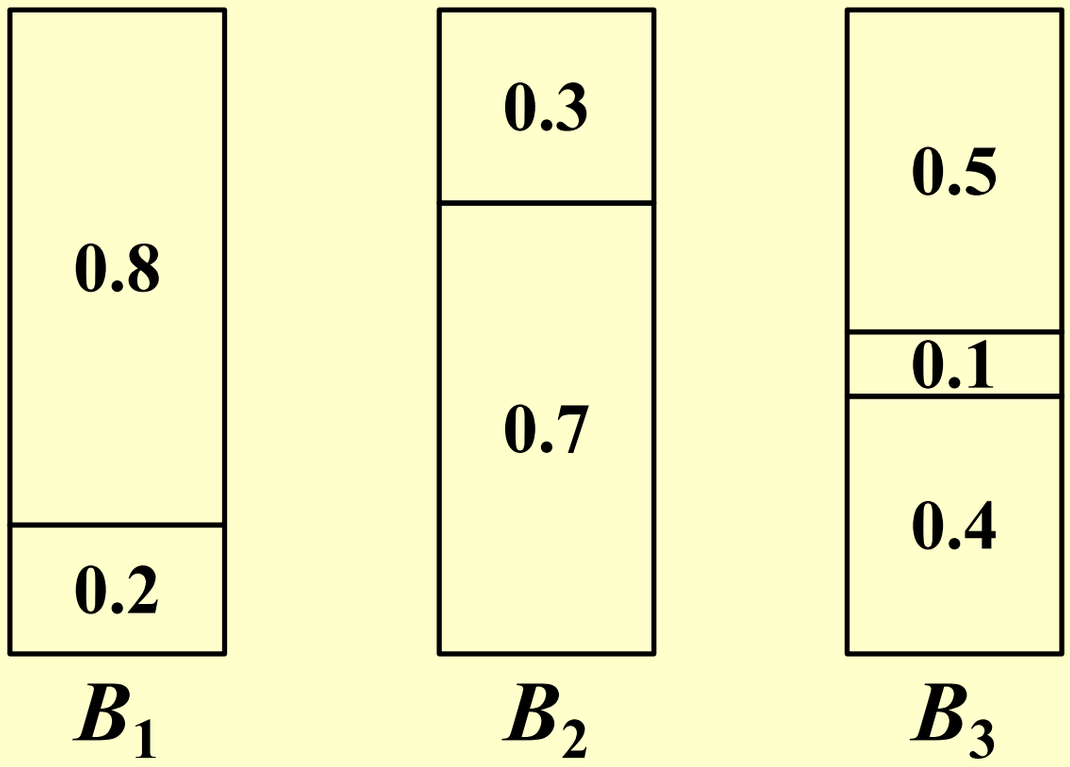



**An Optimal Packing**

# Approximate Bin Packing

**Given $N$ items of sizes $S_1, S_2, \ldots, S_N$, such that $0 < S_i \le 1$ for all $1 \le i \le N$. Pack these items in the fewest number of bins, each of which has unit capacity.**

**〖Example〗** $N = 7$; $S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8$

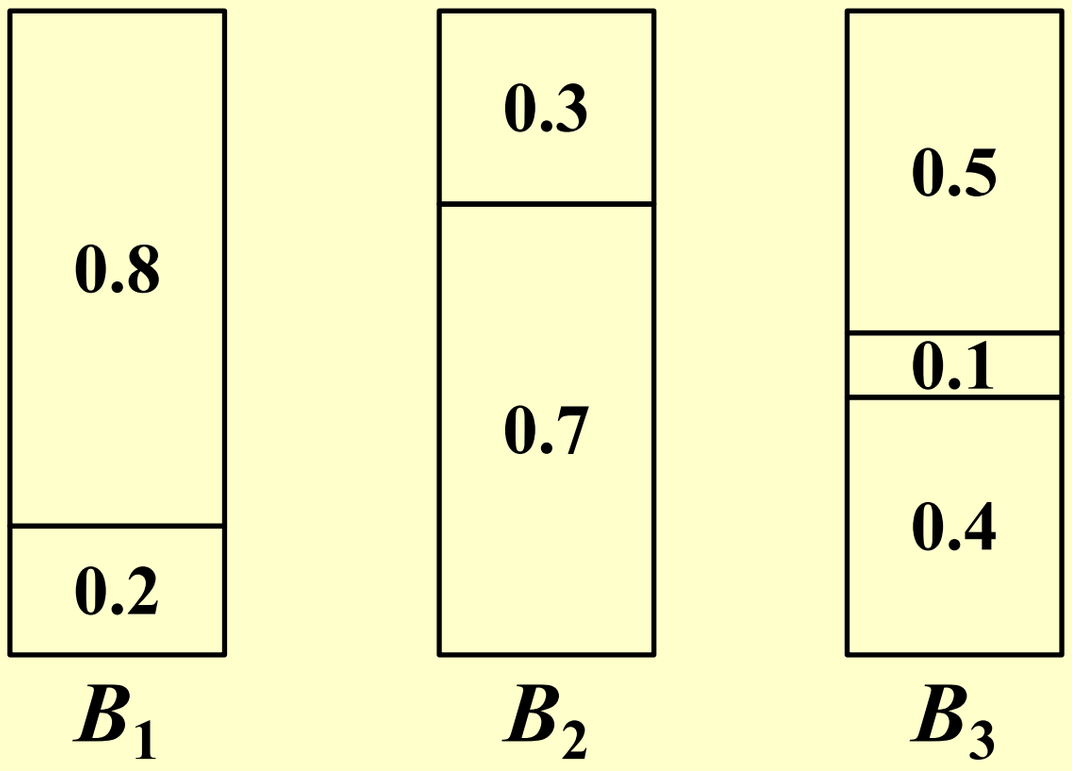



**An Optimal Packing**

# Approximate Bin Packing

**Given $N$ items of sizes $S_1, S_2, \ldots, S_N$, such that $0 < S_i \le 1$ for all $1 \le i \le N$. Pack these items in the fewest number of bins, each of which has unit capacity.**

**〖Example〗** $N = 7$; $S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8$



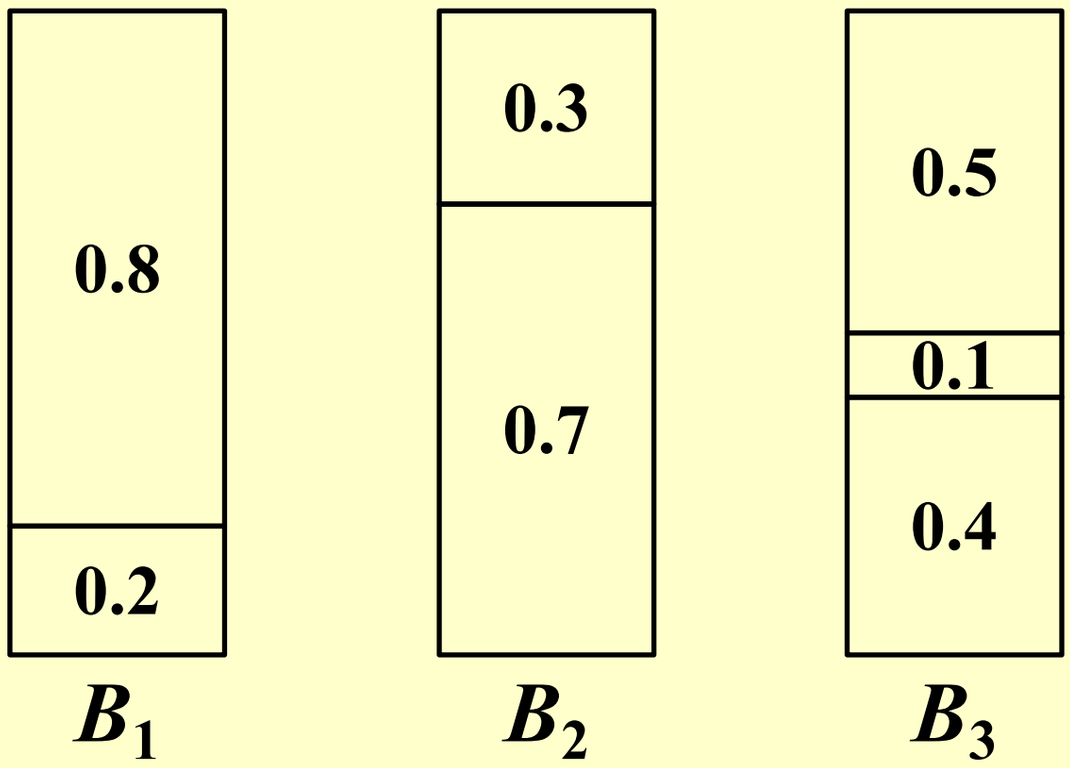

An Optimal Packing

*NP Hard*

# Approximate Bin Packing

**Given $N$ items of sizes $S_1, S_2, \ldots, S_N$, such that $0 < S_i \le 1$ for all $1 \le i \le N$. Pack these items in the fewest number of bins, each of which has unit capacity.**

**〖Example〗** $N = 7$; $S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8$




**An Optimal Packing**

***NP Hard***

The NP-complete 2-partition problem can be reduced to bin packing.
See Slides Page 12.

- **Next Fit**

## ❖ Next Fit

```
void NextFit ( )
{   read item1;
    while ( read item2 ) {
        if ( item2 can be packed in the same bin as item1 )
                place item2 in the bin;
        else
                create a new bin for item2;
        item1 = item2;
    } /* end-while */
}
```

## ❖ Next Fit

```
void NextFit ( )
{   read item1;
    while ( read item2 ) {
        if ( item2 can be packed in the same bin as item1 )
                place item2 in the bin;
        else
                create a new bin for item2;
        item1 = item2;
    } /* end-while */
}
```

**【Theorem】** Let $M$ be the optimal number of bins required to pack a list $I$ of items. Then *next fit* never uses more than $2M - 1$ bins. There exist sequences such that *next fit* uses $2M - 1$ bins.

# A simple proof for Next Fit:

## A simple proof for Next Fit:

**If Next Fit generates $2M$ (or $2M+1$) bins, then the optimal solution must generate at least $M+1$ bins.**

## A simple proof for Next Fit:

**If Next Fit generates $2M$ (or $2M+1$) bins, then the optimal solution must generate at least $M+1$ bins.**

**Let $S(B_i)$ be the size of the $i$th bin. Then we must have:**

# A simple proof for Next Fit:

**If Next Fit generates $2M$ (or $2M+1$) bins, then the optimal solution must generate at least $M+1$ bins.**

**Let $S(B_i)$ be the size of the $i$th bin. Then we must have:**

$$S(B_1) + S(B_2) > 1$$
$$S(B_3) + S(B_4) > 1$$
$$\cdots\cdots\cdots\cdots$$
$$S(B_{2M-1}) + S(B_{2M}) > 1$$

# A simple proof for Next Fit:

**If Next Fit generates $2M$ (or $2M+1$) bins, then the optimal solution must generate at least $M+1$ bins.**

**Let $S(B_i)$ be the size of the $i$th bin. Then we must have:**

$$S(B_1) + S(B_2) > 1$$
$$S(B_3) + S(B_4) > 1$$
$$\cdots\cdots\cdots\cdots$$
$$S(B_{2M-1}) + S(B_{2M}) > 1$$

$$\Rightarrow \quad \sum_{i=1}^{2M} S(B_i) > M$$

# A simple proof for Next Fit:

**If Next Fit generates $2M$ (or $2M+1$) bins, then the optimal solution must generate at least $M+1$ bins.**

**Let $S(B_i)$ be the size of the $i$th bin. Then we must have:**

$$S(B_1) + S(B_2) > 1$$
$$S(B_3) + S(B_4) > 1$$
$$\cdots\cdots\cdots\cdots$$
$$S(B_{2M-1}) + S(B_{2M}) > 1$$

$$\Rightarrow \quad \sum_{i=1}^{2M} S(B_i) > M$$

**The optimal solution needs at least ⌈total size of all the items⌉ bins**

# A simple proof for Next Fit:

**If Next Fit generates $2M$ (or $2M+1$) bins, then the optimal solution must generate at least $M+1$ bins.**

**Let $S(B_i)$ be the size of the $i$th bin. Then we must have:**

$S(B_1) + S(B_2) > 1$

$S(B_3) + S(B_4) > 1$

.....

$S(B_{2M-1}) + S(B_{2M}) > 1$

$$\Rightarrow \quad \sum_{i=1}^{2M} S(B_i) > M$$

**The optimal solution needs at least ⌈total size of all the items⌉ bins**

$$\Rightarrow \lceil \text{total size of all the items} \rceil = \left\lceil \sum_{i=1}^{2M} S(B_i) \right\rceil \geq M + 1$$

- **First Fit**

## ❖ First Fit

```
void FirstFit ( )
{   while ( read item ) {
        scan for the first bin that is large enough for item;
        if ( found )
                place item in that bin;
        else
                create a new bin for item;
    } /* end-while */
}
```

## ❖ First Fit

```
void FirstFit ( )
{   while ( read item ) {
        scan for the first bin that is large enough for item;
        if ( found )
                place item in that bin;
        else
                create a new bin for item;
    } /* end-while */
}
```

Can be implemented in O( $N$ log $N$ )

## ❖ First Fit

```
void FirstFit ( )
{   while ( read item ) {
        scan for the first bin that is large enough for item;
        if ( found )
              place item in that bin;
        else
              create a new bin for item;
    } /* end-while */
}
```

Can be implemented in O( $N$ log $N$ )

【Theorem】 Let $M$ be the optimal number of bins required to pack a list $I$ of items. Then *first fit* never uses more than $17M / 10$ bins. There exist sequences such that *first fit* uses $17(M - 1) / 10$ bins.

## ❖ First Fit

```
void FirstFit ( )
{   while ( read item ) {
        scan for the first bin that is large enough for item;
        if ( found )
              place item in that bin;
        else
              create a new bin for item;
    } /* end-while */
}
```

Can be implemented in O( $N$ log $N$ )

【Theorem】 Let $M$ be the optimal number of bins required to pack a list $I$ of items. Then *first fit* never uses more than $17M / 10$ bins. There exist sequences such that *first fit* uses $17(M - 1) / 10$ bins.

## ❖ Best Fit

Place a new item in the tightest spot among all bins.

## ❖ First Fit

```
void FirstFit ( )
{   while ( read item ) {
        scan for the first bin that is large enough for item;
        if ( found )
              place item in that bin;
        else
              create a new bin for item;
    } /* end-while */
}
```

Can be implemented in O( $N$ log $N$ )

【Theorem】 Let $M$ be the optimal number of bins required to pack a list $I$ of items. Then *first fit* never uses more than $17M / 10$ bins. There exist sequences such that *first fit* uses $17(M - 1) / 10$ bins.

## ❖ Best Fit

Place a new item in the tightest spot among all bins.

$T$ = O( $N$ log $N$ ) and bin no. $\leq 1.7M$

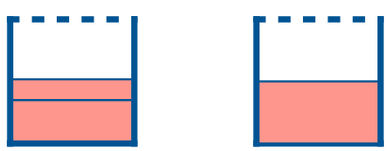
# SOL(First Fit) $\leq$ 2OPT

**Instance**: $n$ items with sizes $s_1, s_2, \ldots, s_n \in (0, 1]$;
*Pack* them into *smallest* number of *unit-sized* bins.

**FirstFit**

Initially $k$=1;
for $i$=1,2, ..., $n$
  item $i$ joins the *first* bin among 1,2, ..., $k$ in which it *fits*;
  if item $i$ can fit into none of these $k$ bins
    open a new bin $k$++ and item i joins it;

Observation: All but at most one bin are more than half full.



$\Rightarrow \sum_i s_i > (SOL - 1) / 2 \Rightarrow SOL - 1 < 2 \sum_i s_i \leq 2\ OPT$

$OPT \geq \sum_i s_i \qquad SOL \leq 2\ OPT$

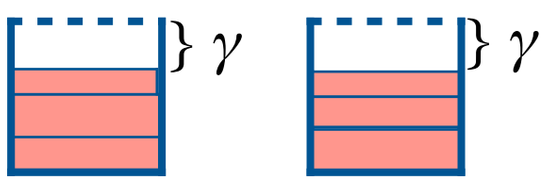
**Instance**: $n$ items with sizes $s_1, s_2, \ldots, s_n \in (0, 1]$;
*Pack* them into *smallest* number of *unit-sized* bins.

**FirstFit**

Initially $k$=1;
for $i$=1,2, ..., $n$
  item $i$ joins the *first* bin among 1,2, ..., $k$ in which it *fits*;
  if item $i$ can fit into none of these $k$ bins
    open a new bin $k$++ and item i joins it;

Assumption: If all items are small, $s_i < \gamma < 0.5$

Observation: All but at most one bin are more than $(1\text{-}\gamma)$ full.




$\Rightarrow \sum_i s_i > (1\text{-}\gamma)(SOL - 1) \Rightarrow SOL \leq OPT / (1\text{-}\gamma) + 1$

**〖Example〗** $S_i$ = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8

〖Example〗 $S_i$ = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8

Next Fit

First Fit

Best Fit

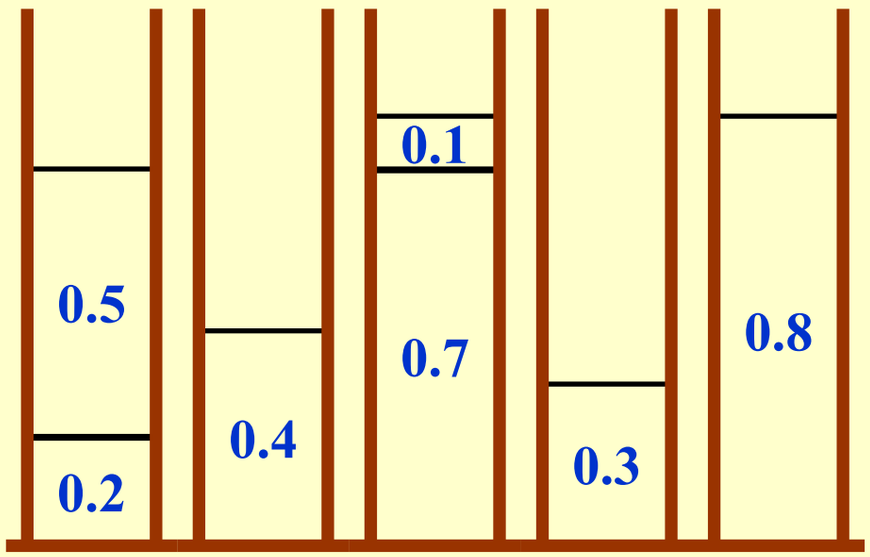
〖Example〗 $S_i = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8$

Next Fit　　First Fit　　Best Fit

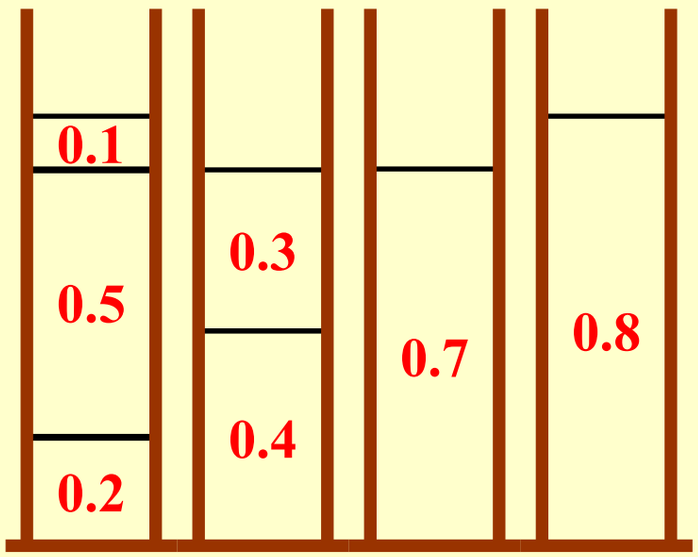
**Discussion 14:** Please show the results.

〖Example〗 $S_i$ = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8

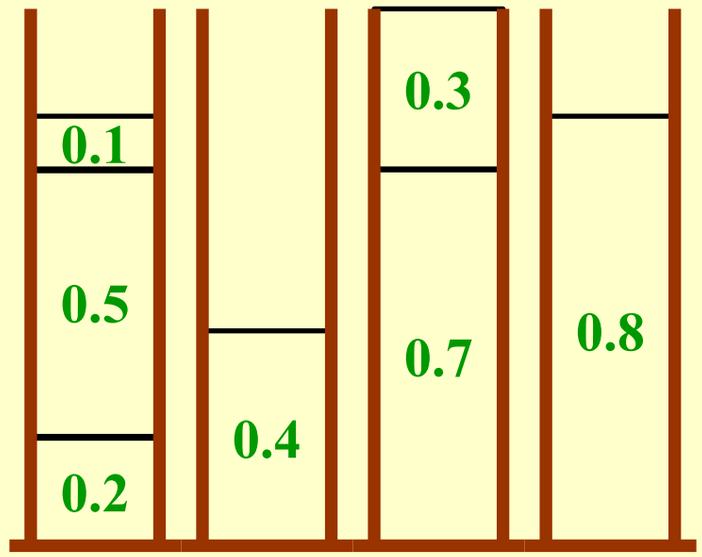
Next Fit




First Fit

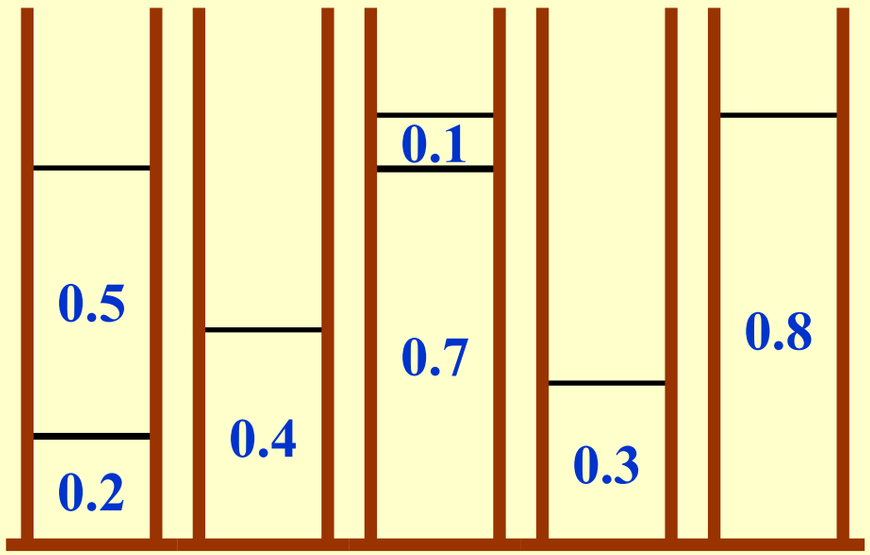



Best Fit

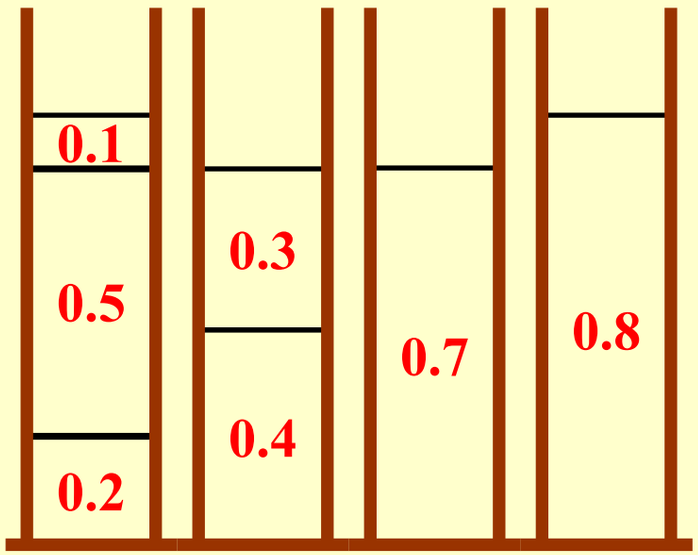


〖Example〗 $S_i$ = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8

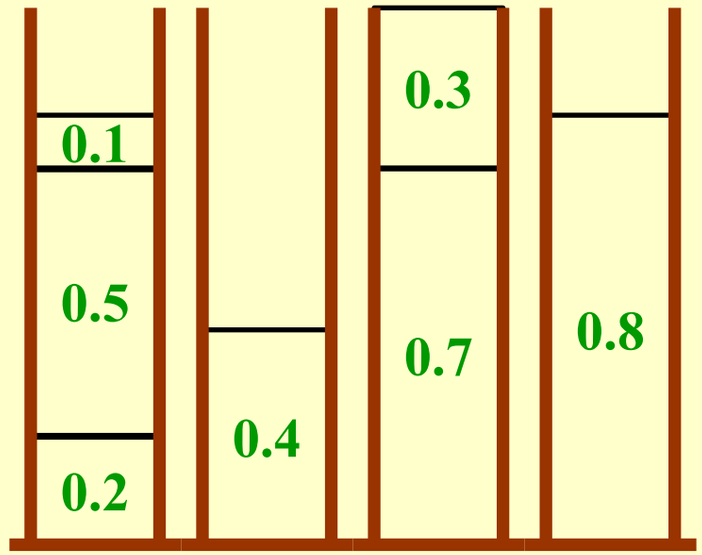
Next Fit




First Fit




Best Fit




〖Example〗 $S_i$ = 1/7+ε, 1/7+ε, 1/7+ε, 1/7+ε, 1/7+ε, 1/7+ε,
1/3+ε, 1/3+ε, 1/3+ε, 1/3+ε, 1/3+ε, 1/3+ε,
1/2+ε, 1/2+ε, 1/2+ε, 1/2+ε, 1/2+ε, 1/2+ε

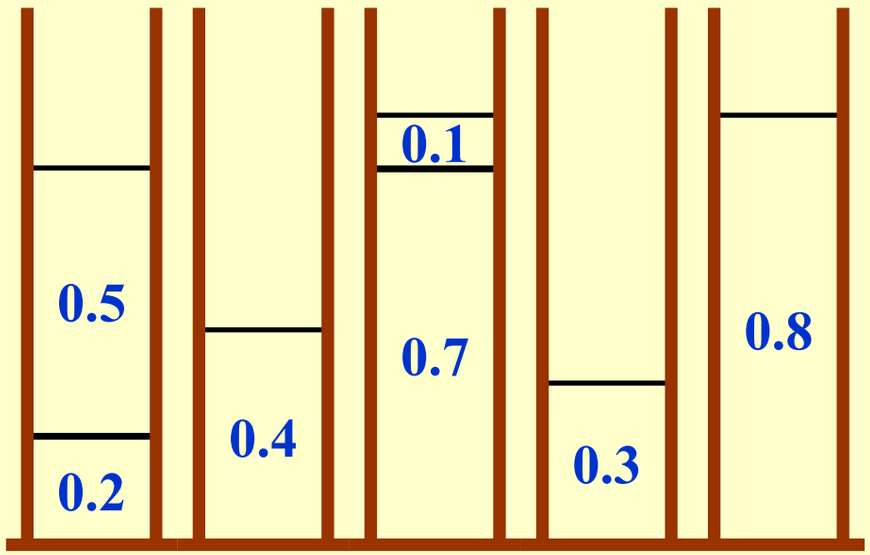
where ε = 0.001.

☞ The optimal solution requires ? bins.

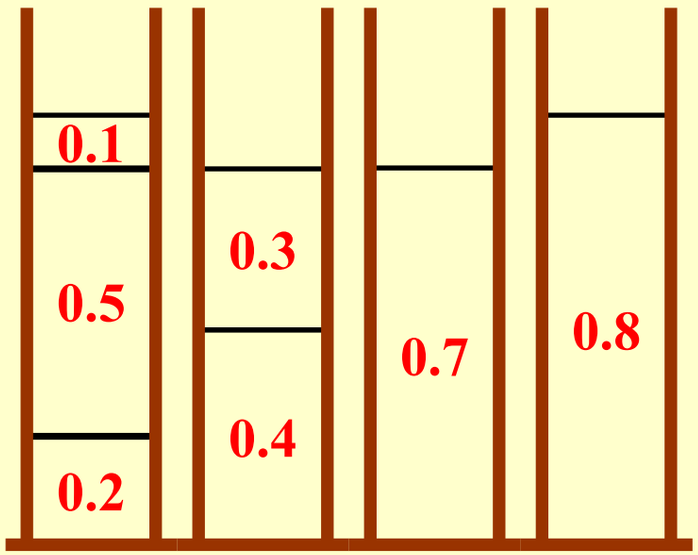
However, all the three on-line algorithms require ? bins.

〖Example〗 $S_i$ = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8

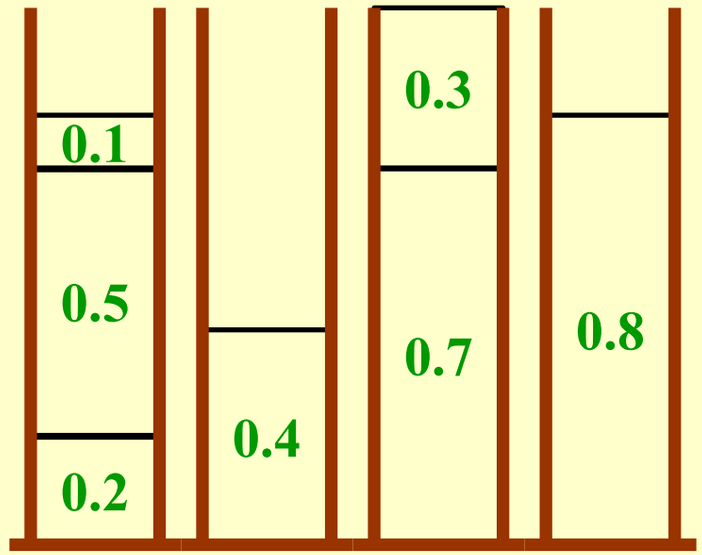
Next Fit




First Fit




Best Fit




〖Example〗 $S_i$ = 1/7+ε, 1/7+ε, 1/7+ε, 1/7+ε, 1/7+ε, 1/7+ε,
1/3+ε, 1/3+ε, 1/3+ε, 1/3+ε, 1/3+ε, 1/3+ε,
1/2+ε, 1/2+ε, 1/2+ε, 1/2+ε, 1/2+ε, 1/2+ε

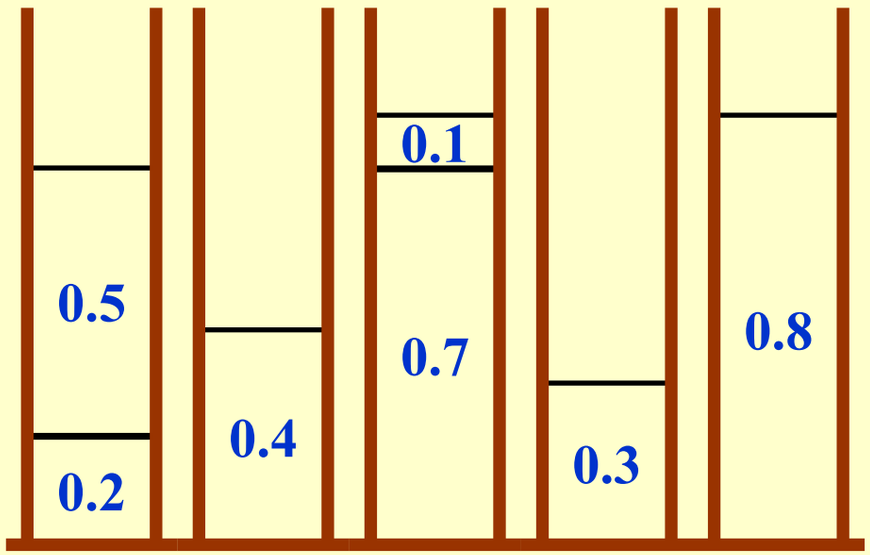
where ε = 0.001.

☞ The optimal solution requires 6 bins.

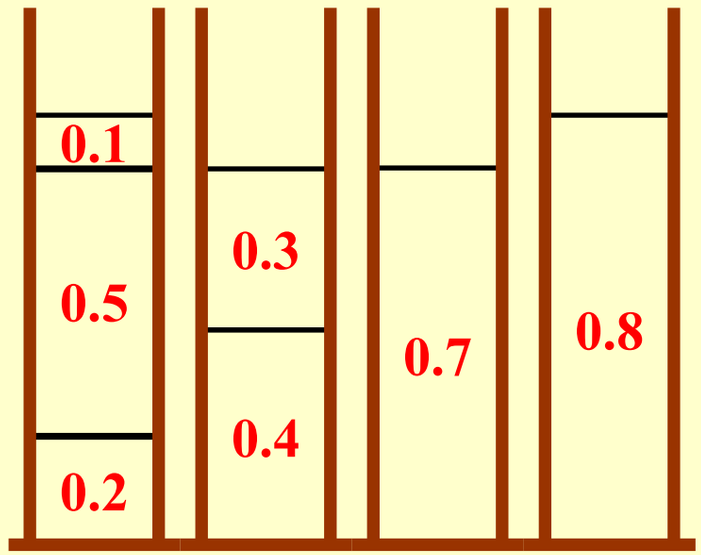
However, all the three on-line algorithms require ? bins.

〖Example〗 $S_i$ = 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8

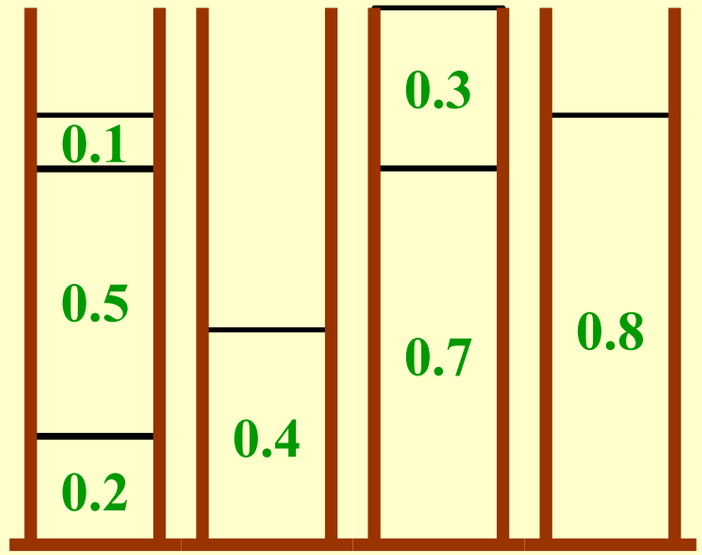
Next Fit




First Fit




Best Fit




〖Example〗 $S_i$ = 1/7+ε, 1/7+ε, 1/7+ε, 1/7+ε, 1/7+ε, 1/7+ε,
1/3+ε, 1/3+ε, 1/3+ε, 1/3+ε, 1/3+ε, 1/3+ε,
1/2+ε, 1/2+ε, 1/2+ε, 1/2+ε, 1/2+ε, 1/2+ε

where ε = 0.001.

☞ The optimal solution requires 6 bins.

However, all the three on-line algorithms require 10 bins.

## ☞ On-line Algorithms

**Place an item before processing the next one, and can NOT change decision.**

☞ **On-line Algorithms**

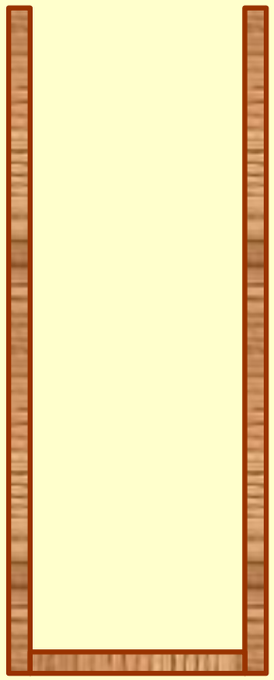
**Place an item before processing the next one, and can NOT change decision.**

**〖Example〗** $S_i = 0.4$

☞ **On-line Algorithms**

**Place an item before processing the next one, and can NOT change decision.**

**〖Example〗** $S_i = 0.4$



## ☞ On-line Algorithms

**Place an item before processing the next one, and can NOT change decision.**

**〖Example〗** $S_i = 0.4$

## ☞ On-line Algorithms

**Place an item before processing the next one, and can NOT change decision.**

**〖Example〗** $S_i = 0.4 , 0.4$

## ☞ On-line Algorithms

**Place an item before processing the next one, and can NOT change decision.**

**〖Example〗** $S_i = 0.4\ ,\ 0.4$

☞ **On-line Algorithms**

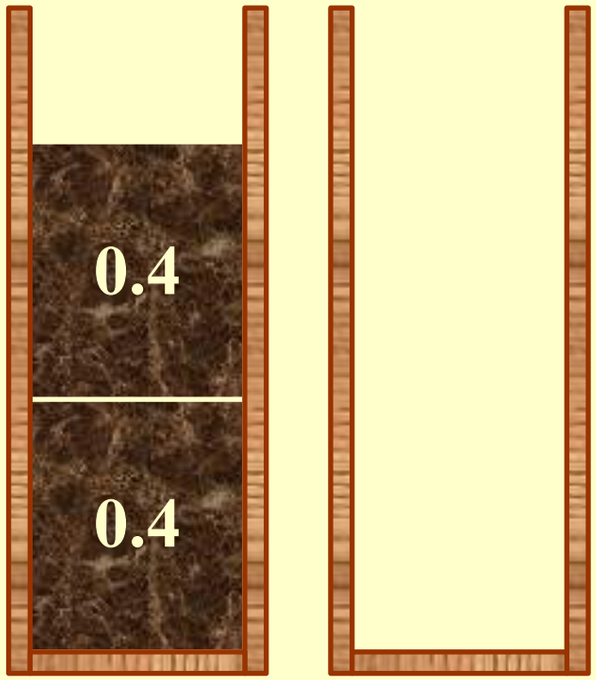
**Place an item before processing the next one, and can NOT change decision.**

**〖Example〗** $S_i = 0.4\ ,\ 0.4,\ 0.6$

☞ **On-line Algorithms**

**Place an item before processing the next one, and can NOT change decision.**

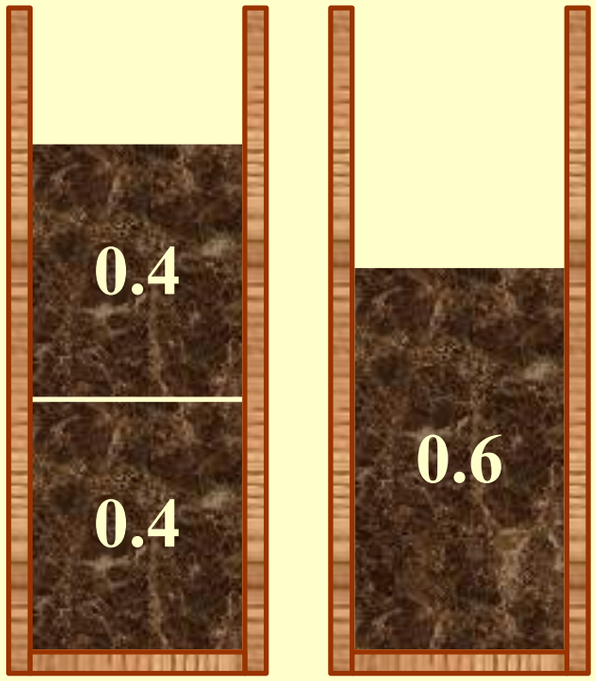
**〖Example〗** $S_i = 0.4\ ,\ 0.4,\ 0.6$



☞ **On-line Algorithms**

**Place an item before processing the next one, and can NOT change decision.**

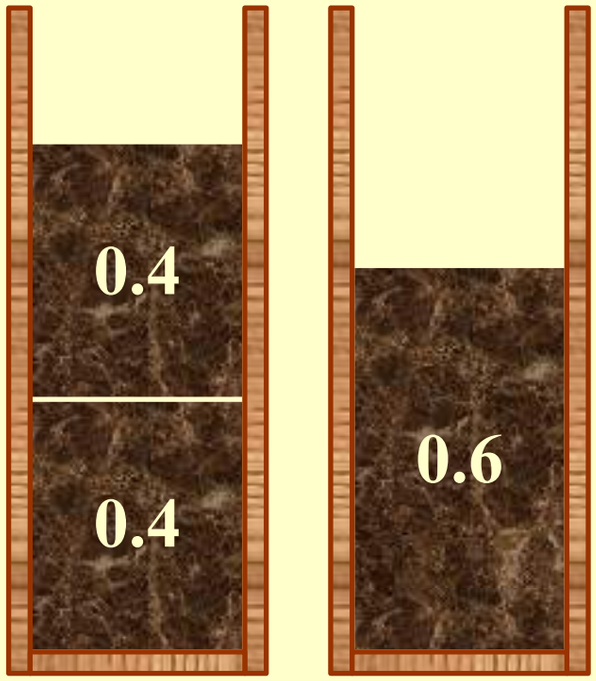
**〖Example〗** $S_i = 0.4\ ,\ 0.4,\ 0.6$



☞ **On-line Algorithms**

**Place an item before processing the next one, and can NOT change decision.**

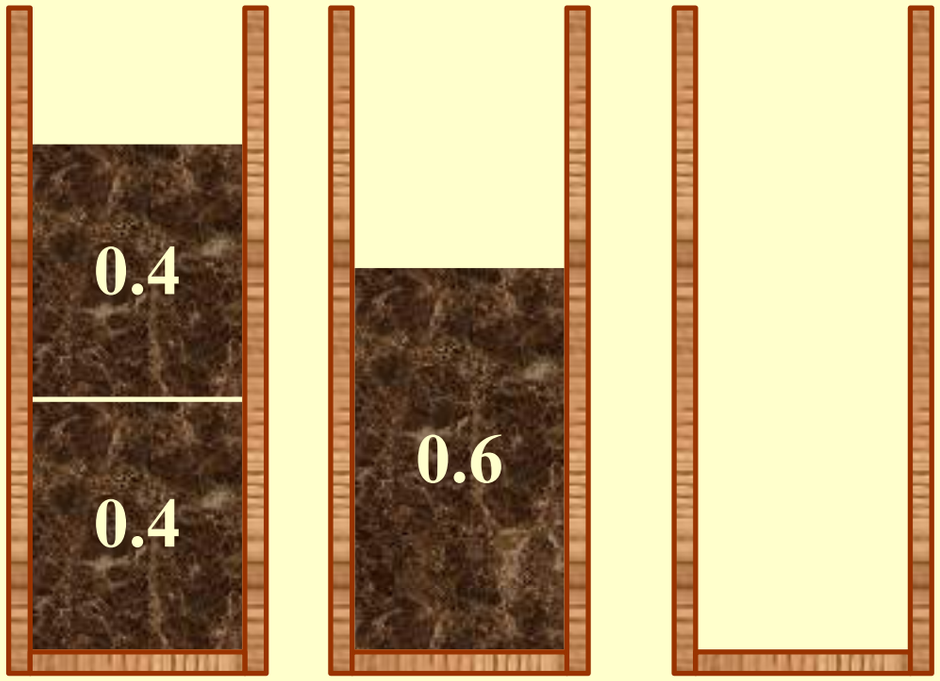
**〖Example〗** $S_i = 0.4\ ,\ 0.4,\ 0.6,\ 0.6$



☞ **On-line Algorithms**

**Place an item before processing the next one, and can NOT change decision.**

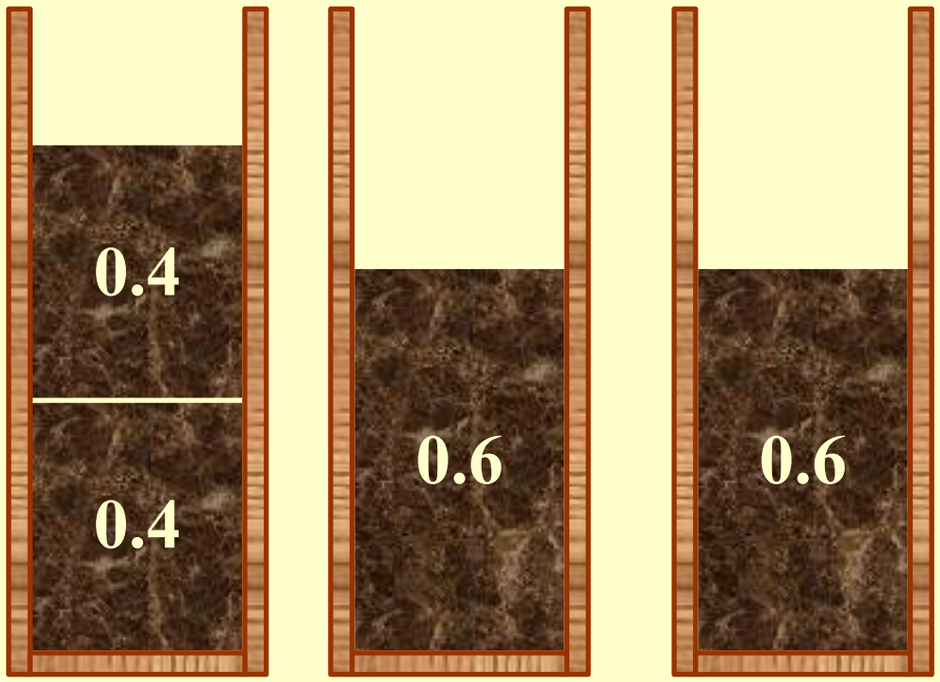
**〖Example〗** $S_i = 0.4\ ,\ 0.4,\ 0.6,\ 0.6$



☞ **On-line Algorithms**

**Place an item before processing the next one, and can NOT change decision.**

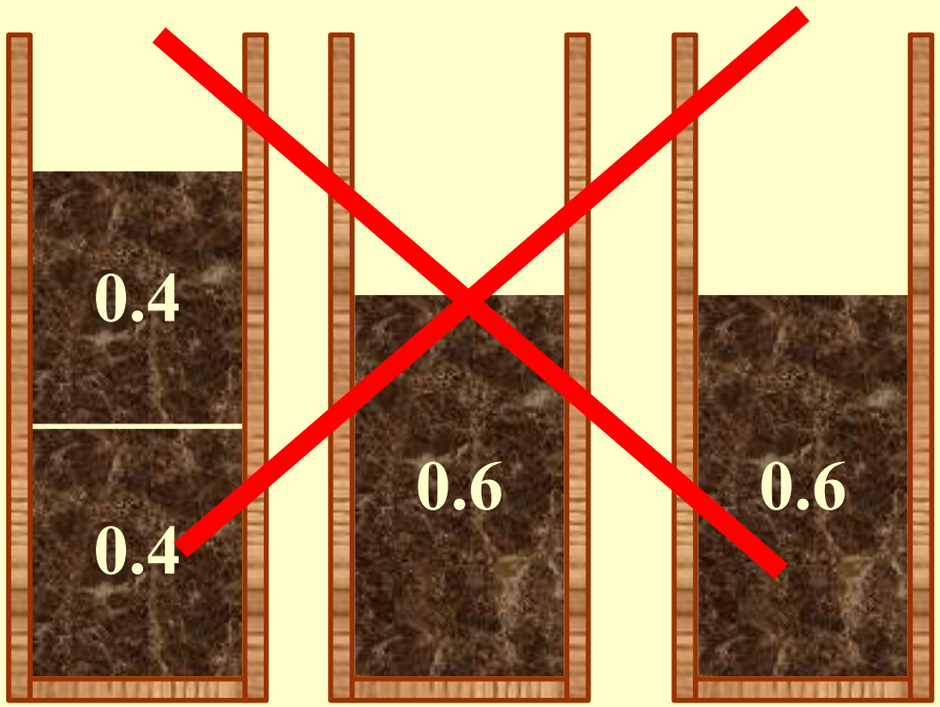
**〖Example〗** $S_i = 0.4\ ,\ 0.4,\ 0.6,\ 0.6$



☞ **On-line Algorithms**

**Place an item before processing the next one, and can NOT change decision.**

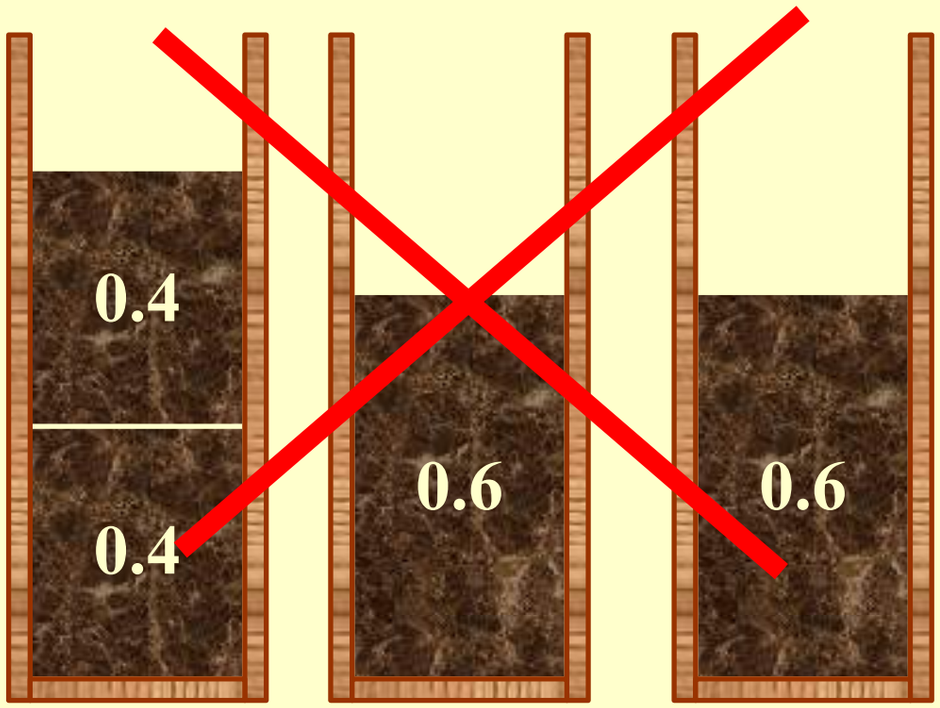
**〖Example〗** $S_i = 0.4\ ,\ 0.4,\ 0.6,\ 0.6$



## ☞ On-line Algorithms

**Place an item before processing the next one, and can NOT change decision.**

**〖Example〗** $S_i = 0.4\ ,\ 0.4,\ 0.6,\ 0.6$

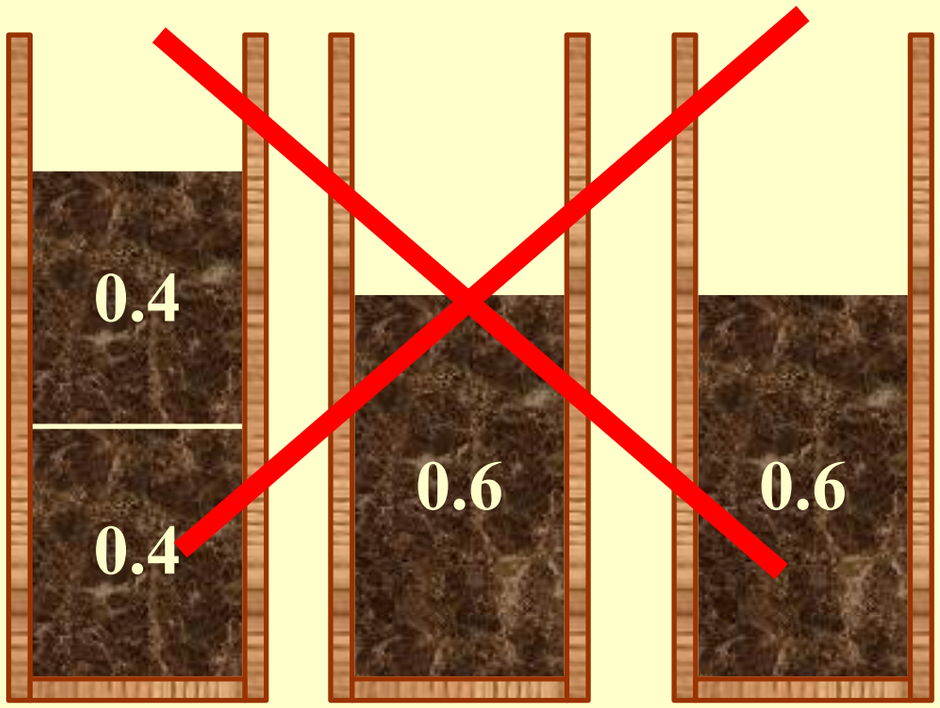



**You never know when the input might end. No on-line algorithm can always give an optimal solution.**

## ☞ On-line Algorithms

**Place an item before processing the next one, and can NOT change decision.**

**〖Example〗** $S_i = 0.4\ ,\ 0.4,\ 0.6,\ 0.6$




**You never know when the input might end. No on-line algorithm can always give an optimal solution.**

**【Theorem】 There are inputs that force any on-line bin-packing algorithm to use at least 5/3 the optimal number of bins.**

# Lower Bound

**Instance**: $n$ items with sizes $s_1, s_2, \ldots, s_n \in (0, 1]$;
*Pack* them into *smallest* number of *unit-sized* bins.

**Theorem**

Unless **P=NP**, there is no poly-time approximation algorithm for bin packing with approximation ratio $<3/2$.

reduction from the partition problem:

**Input**: $n$ numbers $x_1, x_2, \ldots, x_n \in \mathbb{Z}^+$.

Determine whether $\exists$ a partition of $\{1, 2, \ldots, n\}$ into $A$ and $B$ such that $\sum_{i \in A} x_i = \sum_{i \in B} x_i$.

reduction → $n$ items with sizes $s_1, s_2, \ldots, s_n$ where $s_i = 2x_i / \sum_j x_j$

$\exists$ a packing into 2 unit-sized bins → "yes" ↘ partition

all packings use ≥3 unit-sized bins → "no" ↗ problem

**Instance**: $n$ items with sizes $s_1, s_2, \ldots, s_n \in (0, 1]$;
*Pack* them into *smallest* number of *unit-sized* bins.

**Theorem**

Unless **P=NP**, there is no poly-time approximation algorithm for bin packing with approximation ratio $<3/2$.

It is **NP**-hard to distinguish between:

- the instances with OPT = 2;
- the instances with OPT ≥ 3.

**FirstFitDecreasing (FFD)**

Sort items in non-increasing order of sizes;
run **FirstFit**;

FFD returns a packing into $\leq 11/9\ OPT + 6/9$ bins

## ☞ Off-line Algorithms

**View the entire item list before producing an answer.**

☞ **Off-line Algorithms**

**View the entire item list before producing an answer.**

**Trouble-maker: The large items**

## ☞ Off-line Algorithms

View the entire item list before producing an answer.

**Trouble-maker**: The large items

**Solution**: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.

☞ **Off-line Algorithms**

**View the entire item list before producing an answer.**

**Trouble-maker: The large items**

**Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.**

**〖Example〗** $S_i$ **= 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8**

☞ **Off-line Algorithms**

**View the entire item list before producing an answer.**

**Trouble-maker: The large items**

**Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.**

**〖Example〗** $S_i$ = **0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1**

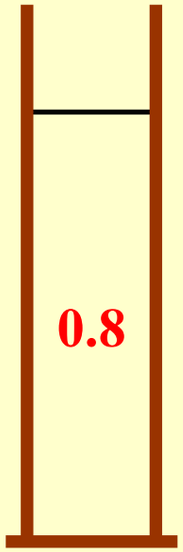
## ☞ Off-line Algorithms

**View the entire item list before producing an answer.**

**Trouble-maker: The large items**

**Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.**

**〖Example〗** $S_i$ = 0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1



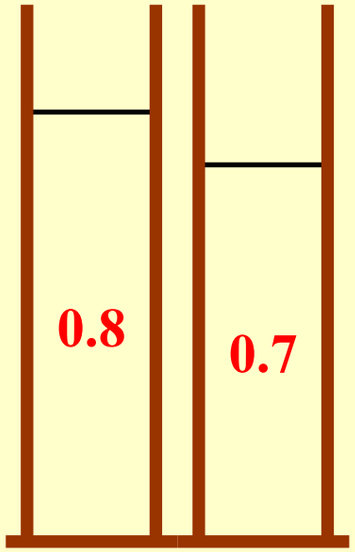
☞ **Off-line Algorithms**

**View the entire item list before producing an answer.**

**Trouble-maker: The large items**

**Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.**

**〖Example〗** $S_i$ = **0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1**



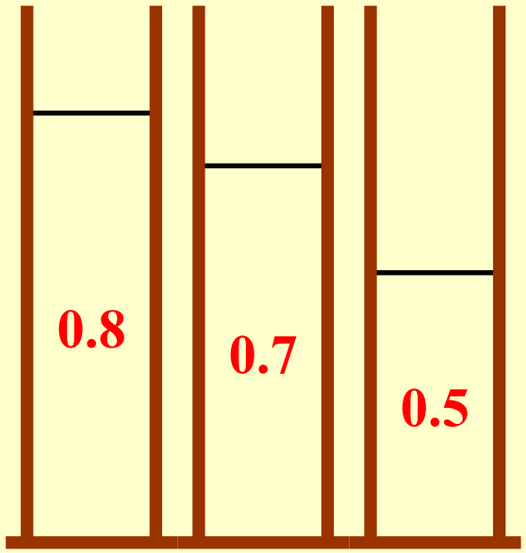
☞ **Off-line Algorithms**

**View the entire item list before producing an answer.**

**Trouble-maker: The large items**

**Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.**

**〚Example〛** $S_i$ = **0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1**



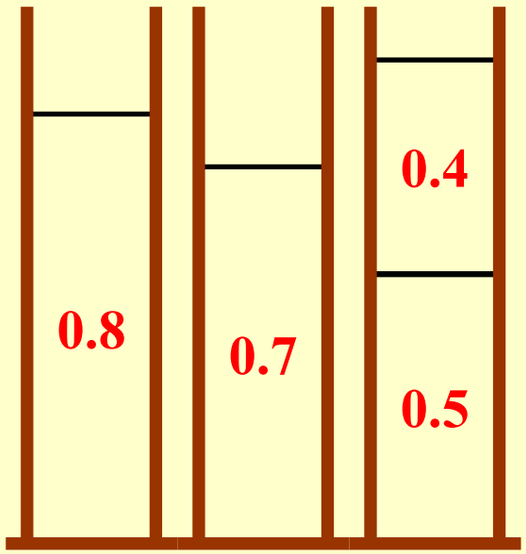
## ☞ Off-line Algorithms

**View the entire item list before producing an answer.**

**Trouble-maker: The large items**

**Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.**

**〖Example〗** $S_i$ = 0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1



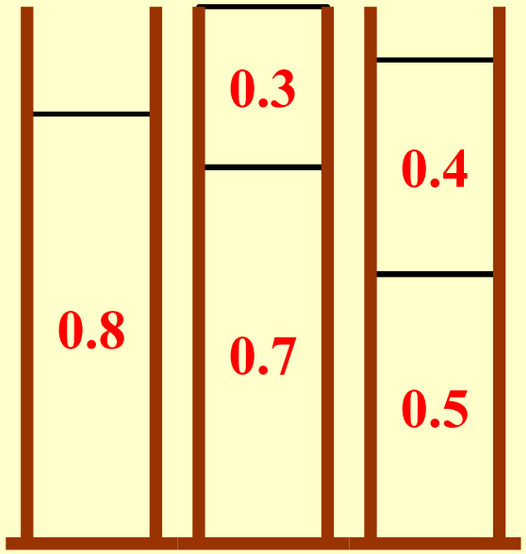
## ☞ Off-line Algorithms

**View the entire item list before producing an answer.**

**Trouble-maker: The large items**

**Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.**

**〖Example〗** $S_i$ = 0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1



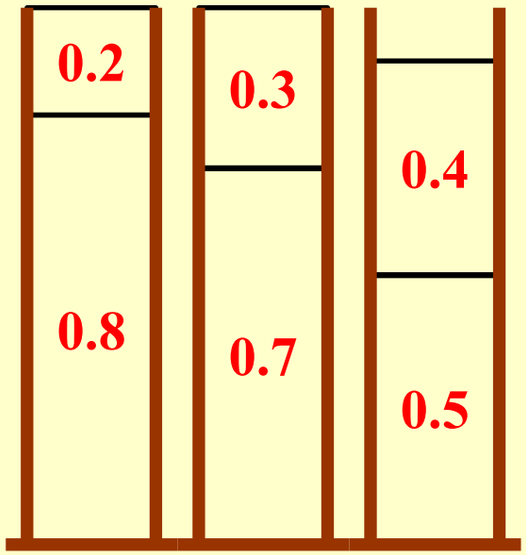
## ☞ Off-line Algorithms

**View the entire item list before producing an answer.**

**Trouble-maker: The large items**

**Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.**

**〖Example〗** $S_i$ = 0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1



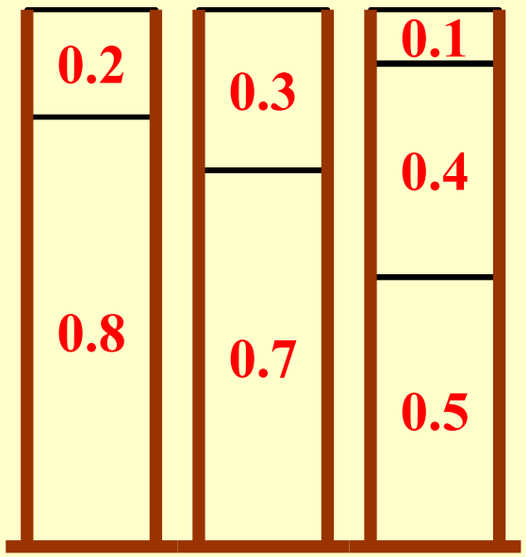
## ☞ Off-line Algorithms

**View the entire item list before producing an answer.**

**Trouble-maker: The large items**

**Solution: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.**

**〖Example〗** $S_i$ = 0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1



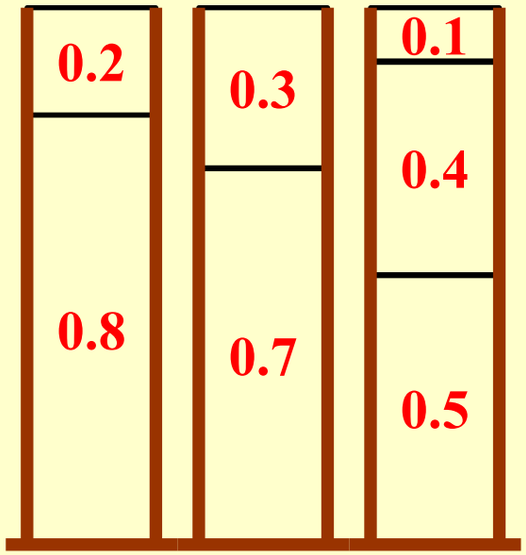
## ☞ Off-line Algorithms

View the entire item list before producing an answer.

**Trouble-maker**: The large items

**Solution**: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.

〖Example〗 $S_i$ = 0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1




【Theorem】 Let $M$ be the optimal number of bins required to pack a list $I$ of items. Then *first fit decreasing* never uses more than $11M / 9 + 6/9$ bins. There exist sequences such that *first fit decreasing* uses $11M / 9 + 6/9$ bins.

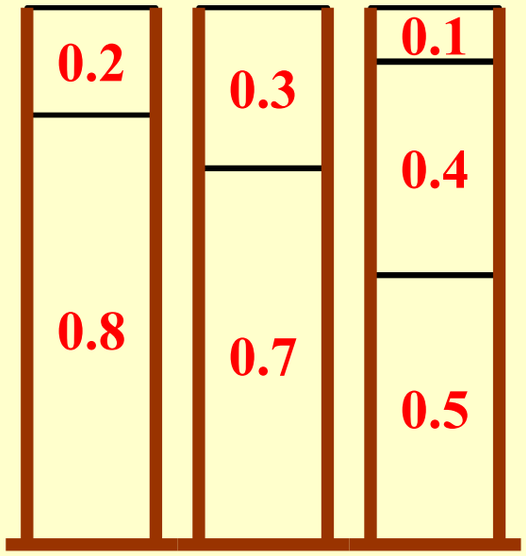
## ☞ Off-line Algorithms

View the entire item list before producing an answer.

**Trouble-maker**: The large items

**Solution**: Sort the items into non-increasing sequence of sizes. Then apply first (or best) fit – *first* (or *best*) *fit decreasing*.

〖Example〗 $S_i$ = 0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1




【Theorem】 Let $M$ be the optimal number of bins required to pack a list $I$ of items. Then *first fit decreasing* never uses more than $11M / 9 + 6/9$ bins. There exist sequences such that *first fit decreasing* uses $11M / 9 + 6/9$ bins.

**Simple greedy heuristics can give good results.**

# Approximation

Optimization problem $\Pi$: minimization/maximization

- a set $D$ of valid instances (inputs);
- each instance $I \in D$ defines a set of feasible solutions $S(I)$;
- an objective function $obj$ that assigns each instance $I \in D$ and solution $s \in S(I)$ a value.

$$\mathrm{OPT}(I) = \text{objective value of optimal solution } s^* \in S(I) \text{ of instance } I$$

- algorithm $A$: returns a solution $s \in S(I)$ on every instance $I$

$$\mathrm{SOL}_A(I) = \text{objective value of the solution } s \in S(I) \text{ returned by } A \text{ on instance } I$$

minimization: approximation ratio of algorithm $A$ is $\alpha$

$$\text{if } \forall \text{ instance } I: \quad \frac{\mathrm{SOL}_A(I)}{\mathrm{OPT}(I)} \leq \alpha$$

# Approximation

Optimization problem $\Pi$: minimization/maximization

- a set $D$ of valid instances (inputs);
- each instance $I \in D$ defines a set of feasible solutions $S(I)$;
- an objective function $obj$ that assigns each instance $I \in D$ and solution $s \in S(I)$ a value.

$$\mathrm{OPT}(I) = \text{objective value of optimal solution } s^* \in S(I) \text{ of instance } I$$

- algorithm $A$: returns a solution $s \in S(I)$ on every instance $I$

$$\mathrm{SOL}_A(I) = \text{objective value of the solution } s \in S(I) \text{ returned by } A \text{ on instance } I$$

maximization: approximation ratio of algorithm $A$ is $\alpha$

$$\text{if } \forall \text{ instance } I: \quad \frac{\mathrm{SOL}_A(I)}{\mathrm{OPT}(I)} \geq \alpha$$

# Outline: Approximation Algorithms

- Bin packing
- 0-1 Knapsack
- K-center selection
- Take-home messages

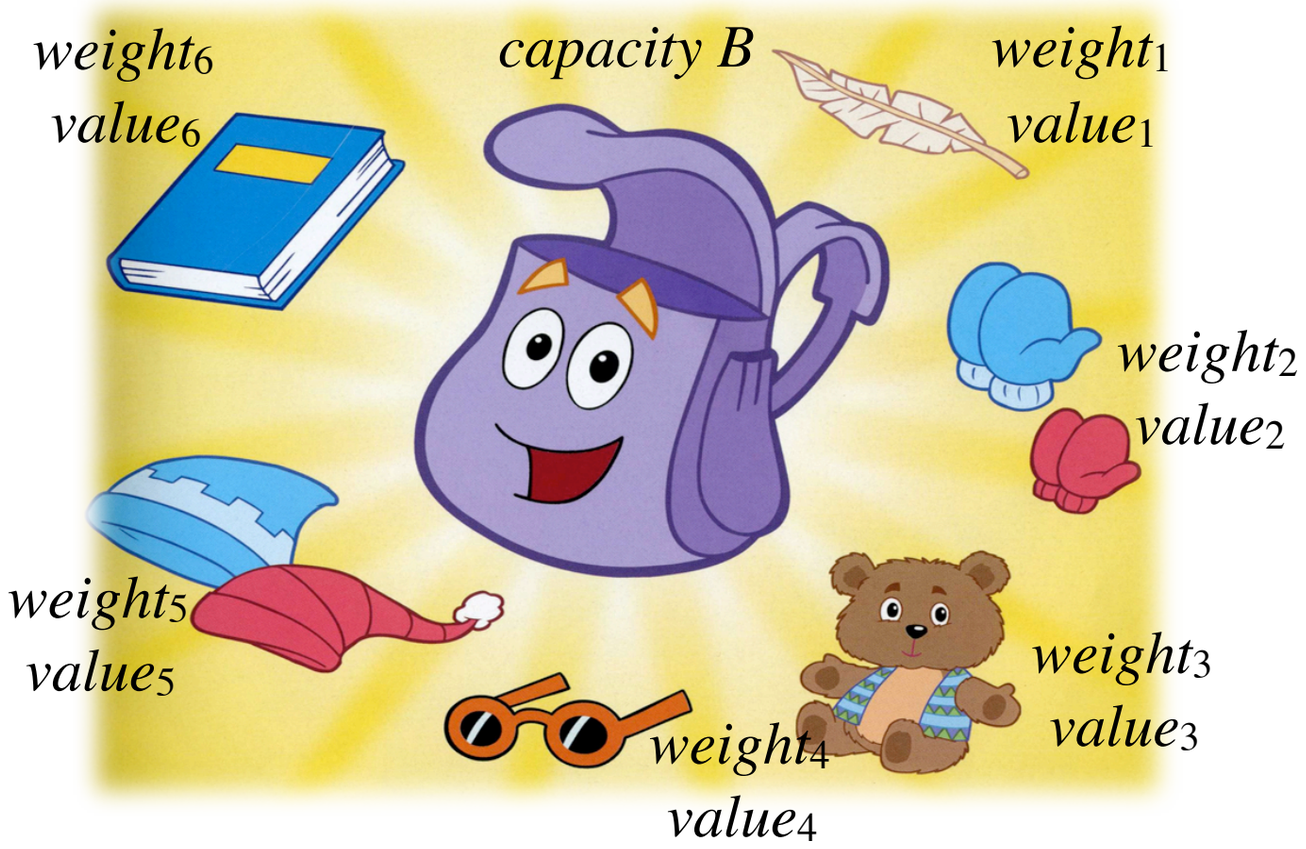
# *Knapsack* Problem

**Instance**: $n$ items $i$=1,2, ..., $n$;

weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;

knapsack capacity $B \in \mathbb{Z}^+$;

Find a subset of items whose total weight is bounded by $B$ and total value is maximized.



# *Knapsack* Problem

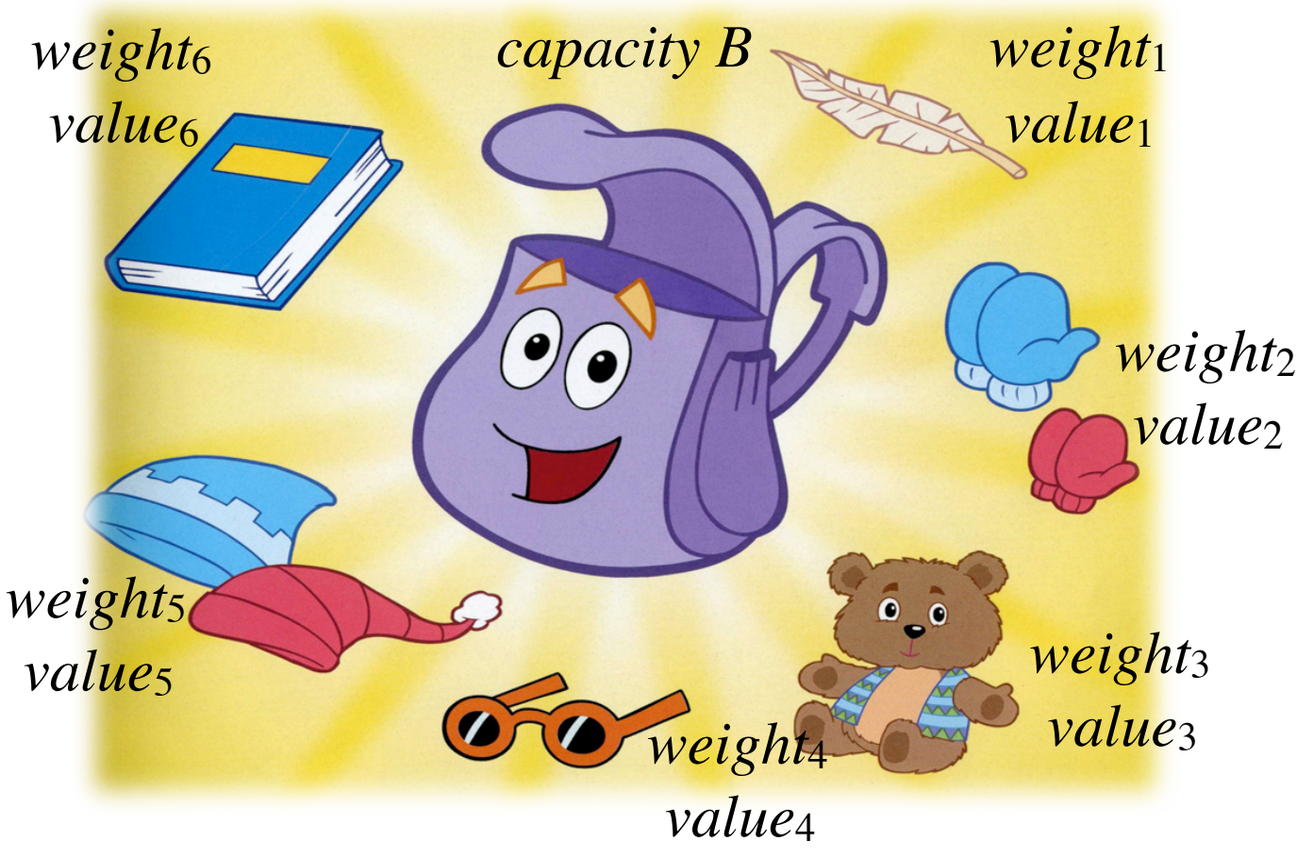
**Instance**: $n$ items $i$=1,2, ..., $n$;

weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;

knapsack capacity $B \in \mathbb{Z}^+$;

Find a subset of items whose total weight is bounded by $B$ and total value is maximized.

- 0-1 Knapsack problem
- one of Karp's 21 **NP**-complete problems



# Knapsack is NP-complete

KNAPSACK. Given a set $X$, weights $w_i \geq 0$, values $v_i \geq 0$, a weight limit $W$, and a target value $V$, is there a subset $S \subseteq X$ such that:

$$\begin{aligned} \sum_{i \in S} w_i &\leq W \\ \sum_{i \in S} v_i &\geq V \end{aligned}$$

SUBSET-SUM. Given a set $X$, values $u_i \geq 0$, and an integer $U$, is there a subset $S \subseteq X$ whose elements sum to exactly $U$?

Theorem. SUBSET-SUM $\leq_P$ KNAPSACK.

Pf. Given instance $(u_1, \ldots, u_n, U)$ of SUBSET-SUM, create KNAPSACK instance:

$$\begin{aligned} v_i = w_i = u_i \qquad & \sum_{i \in S} u_i \leq U \\ V = W = U \qquad & \sum_{i \in S} u_i \geq U \end{aligned}$$

# Greedy *Heuristics*

**Instance**: $n$ items $i$=1,2, ..., $n$;

weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;

knapsack capacity $B \in \mathbb{Z}^+$;

Find an $S \subseteq \{1,2, \ldots, n\}$ that maximizes $\sum_{i \in S} v_i$

subject to $\sum_{i \in S} w_i \leq B$.

Sort all items according to the ratio $r_i = v_i/w_i$

so that $r_1 \geq r_2 \geq \cdots \geq r_n$;

for $i$=1,2, ..., $n$

item $i$ joins $S$ if the resulting total weight $\leq B$;

approximation ratio: arbitrarily bad

# Greedy *Heuristics*

**Instance**: $n$ items $i$=1,2, ..., $n$;

weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;

knapsack capacity $B \in \mathbb{Z}^+$;

Find an $S \subseteq \{1,2, \ldots, n\}$ that maximizes $\sum_{i \in S} v_i$

subject to $\sum_{i \in S} w_i \leq B$.

Sort all items according to the ratio $r_i = v_i/w_i$

so that $r_1 \geq r_2 \geq \cdots \geq r_n$;

for $i$=1,2, ..., $n$

item $i$ joins $S$ if the resulting total weight $\leq B$;

approximation ratio: arbitrarily bad

Interestingly, we can have a 2-approximation algorithm with greedy technique.

# The Knapsack Problem — fractional version

# The Knapsack Problem — fractional version

A knapsack with a capacity $M$ is to be packed. Given $N$ items. Each item $i$ has a weight $w_i$ and a profit $p_i$. If $x_i$ is the percentage of the item $i$ being packed, then the packed profit will be $p_i x_i$.

# The Knapsack Problem — fractional version

A knapsack with a capacity $M$ is to be packed. Given $N$ items. Each item $i$ has a weight $w_i$ and a profit $p_i$. If $x_i$ is the percentage of the item $i$ being packed, then the packed profit will be $p_i x_i$.

An optimal packing is a feasible one with maximum profit. That is, we are supposed to find the values of $x_i$ such that $\sum_{i=1}^{n} p_i x_i$ obtains its maximum under the constraints

$$\sum_{i=1}^{n} w_i x_i \le M \quad \text{and} \quad x_i \in [0,1] \text{ for } 1 \le i \le n$$

# The Knapsack Problem — fractional version

A knapsack with a capacity $M$ is to be packed. Given $N$ items. Each item $i$ has a weight $w_i$ and a profit $p_i$. If $x_i$ is the percentage of the item $i$ being packed, then the packed profit will be $p_i x_i$.

An optimal packing is a feasible one with maximum profit. That is, we are supposed to find the values of $x_i$ such that $\sum_{i=1}^{n} p_i x_i$ obtains its maximum under the constraints

$$\sum_{i=1}^{n} w_i x_i \le M \quad \text{and} \quad x_i \in [0,1] \text{ for } 1 \le i \le n$$

**Q:** What must we do in each stage?

# The Knapsack Problem — fractional version

A knapsack with a capacity $M$ is to be packed. Given $N$ items. Each item $i$ has a weight $w_i$ and a profit $p_i$. If $x_i$ is the percentage of the item $i$ being packed, then the packed profit will be $p_i x_i$.

An optimal packing is a feasible one with maximum profit. That is, we are supposed to find the values of $x_i$ such that $\sum_{i=1}^{n} p_i x_i$ obtains its maximum under the constraints

$$\sum_{i=1}^{n} w_i x_i \le M \quad \text{and} \quad x_i \in [0,1] \text{ for } 1 \le i \le n$$

**Q:** What must we do in each stage?

**A:** Pack one item into the knapsack.

# The Knapsack Problem — fractional version

A knapsack with a capacity $M$ is to be packed. Given $N$ items. Each item $i$ has a weight $w_i$ and a profit $p_i$. If $x_i$ is the percentage of the item $i$ being packed, then the packed profit will be $p_i x_i$.

An optimal packing is a feasible one with maximum profit. That is, we are supposed to find the values of $x_i$ such that $\sum_{i=1}^{n} p_i x_i$ obtains its maximum under the constraints

$$\sum_{i=1}^{n} w_i x_i \le M \quad \text{and} \quad x_i \in [0,1] \text{ for } 1 \le i \le n$$

**Q:** What must we do in each stage?

**A:** Pack one item into the knapsack.

**Q:** On which criterion shall we be greedy?

# The Knapsack Problem — fractional version

A knapsack with a capacity $M$ is to be packed. Given $N$ items. Each item $i$ has a weight $w_i$ and a profit $p_i$. If $x_i$ is the percentage of the item $i$ being packed, then the packed profit will be $p_i x_i$.

An optimal packing is a feasible one with maximum profit. That is, we are supposed to find the values of $x_i$ such that $\sum_{i=1}^{n} p_i x_i$ obtains its maximum under the constraints

$$\sum_{i=1}^{n} w_i x_i \le M \quad \text{and} \quad x_i \in [0,1] \text{ for } 1 \le i \le n$$

**Q:** What must we do in each stage?

**A:** Pack one item into the knapsack.

**Q:** On which criterion shall we be greedy?

① maximum profit

② minimum weight

③ maximum profit density $p_i / w_i$

# The Knapsack Problem — fractional version

A knapsack with a capacity $M$ is to be packed. Given $N$ items. Each item $i$ has a weight $w_i$ and a profit $p_i$. If $x_i$ is the percentage of the item $i$ being packed, then the packed profit will be $p_i x_i$.

An optimal packing is a feasible one with maximum profit. That is, we are supposed to find the values of $x_i$ such that $\sum_{i=1}^{n} p_i x_i$ obtains its maximum under the constraints

$$\sum_{i=1}^{n} w_i x_i \le M \quad \text{and} \quad x_i \in [0,1] \text{ for } 1 \le i \le n$$

**Q:** What must we do in each stage?

**A:** Pack one item into the knapsack.

**Q:** On which criterion shall we be greedy?

① maximum profit

② minimum weight

③ maximum profit density $p_i / w_i$

# The Knapsack Problem — fractional version

A knapsack with a capacity $M$ is to be packed. Given $N$ items. Each item $i$ has a weight $w_i$ and a profit $p_i$. If $x_i$ is the percentage of the item $i$ being packed, then the packed profit will be $p_i x_i$.

An optimal packing is a feasible one with maximum profit. That is, we are supposed to find the values of $x_i$ such that $\sum_{i=1}^{n} p_i x_i$ obtains its maximum under the constraints

$$\sum_{i=1}^{n} w_i x_i \le M \quad \text{and} \quad x_i \in [0,1] \text{ for } 1 \le i \le n$$

**Q:** What must we do in each stage?

**A:** Pack one item into the knapsack.

**Q:** On which criterion shall we be greedy?

① maximum profit

② minimum weight

③ maximum profit density $p_i / w_i$

**Example :**

$n = 3$, $M = 20$,

$(p1, p2, p3) = (25, 24, 15)$

$(w1, w2, w3) = (18, 15, 10)$

Solution is...?

# The Knapsack Problem — fractional version

A knapsack with a capacity $M$ is to be packed. Given $N$ items. Each item $i$ has a weight $w_i$ and a profit $p_i$. If $x_i$ is the percentage of the item $i$ being packed, then the packed profit will be $p_i x_i$.

An optimal packing is a feasible one with maximum profit. That is, we are supposed to find the values of $x_i$ such that $\sum_{i=1}^{n} p_i x_i$ obtains its maximum under the constraints

$$\sum_{i=1}^{n} w_i x_i \le M \quad \text{and} \quad x_i \in [0,1] \text{ for } 1 \le i \le n$$

**Q:** What must we do in each stage?

**A:** Pack one item into the knapsack.

**Q:** On which criterion shall we be greedy?

① maximum profit

② minimum weight

③ maximum profit density $p_i / w_i$

**Example :**

$n = 3$, $M = 20$,

$(p1, p2, p3) = (25, 24, 15)$

$(w1, w2, w3) = (18, 15, 10)$

Solution is...?

( 0, 1, 1/2 )

P = 31.5

# The Knapsack Problem — 0-1 version

# The Knapsack Problem — 0-1 version

$x_i$ is either 1 or 0

# The Knapsack Problem — 0-1 version

*NP-hard*

# The Knapsack Problem — 0-1 version

*NP-hard*

**Example :**

$n = 5,\ M = 11,$

$(p_1,\ p_2,\ p_3,\ p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2,\ \ 5,\ \ 6,\ \ 7)$

**Solution is...?**

# The Knapsack Problem — 0-1 version

*NP-hard*

**Example :**

$n = 5,\ M = 11,$

$(p_1,\ p_2,\ p_3,\ p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2,\ \ 5,\ \ 6,\ \ 7)$

Solution is...?

**( 0, 0, 1, 1, 0 )**

**P = 40**

# The Knapsack Problem — 0-1 version

*NP-hard*

**Example :**

$n = 5,\ M = 11,$

$(p_1,\ p_2,\ p_3,\ p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2,\ 5,\ 6,\ 7)$

The greedy solution is

Solution is...?

**( 0, 0, 1, 1, 0 )**

**P = 40**

# The Knapsack Problem — 0-1 version

*NP-hard*

**Example :**

$n = 5,\ M = 11,$

$(p_1,\ p_2,\ p_3,\ p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2,\ 5,\ 6,\ 7)$

Solution is...? **( 0, 0, 1, 1, 0 )**
**P = 40**

The greedy solution is

**( 1, 1, 0, 0, 1 )**
**P = 35**

# The Knapsack Problem — 0-1 version

*NP-hard*

**Example :**

$n = 5,\ M = 11,$

$(p_1,\ p_2,\ p_3,\ p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2,\ 5,\ 6,\ 7)$

Solution is...? **( 0, 0, 1, 1, 0 )**
**P = 40**

The greedy solution is
**( 1, 1, 0, 0, 1 )**
**P = 35**

**What if we run both maximum profit greedy and profit density greedy, then output the better result?**

# The Knapsack Problem — 0-1 version

*NP-hard*

**Example :**

$n = 5,\ M = 11,$

$(p_1,\ p_2,\ p_3,\ p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2,\ 5,\ 6,\ 7)$

Solution is...? ( 0, 0, 1, 1, 0 )
P = 40

The greedy solution is
( 1, 1, 0, 0, 1 )
P = 35

**What if we run both maximum profit greedy and profit density greedy, then output the better result?**

**The approximation ratio is 2.**

# The Knapsack Problem — 0-1 version

*NP-hard*

**Example :**

$n = 5,\ M = 11,$

$(p_1,\ p_2,\ p_3,\ p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2,\ 5,\ 6,\ 7)$

Solution is...? ( 0, 0, 1, 1, 0 )
P = 40

The greedy solution is
( 1, 1, 0, 0, 1 )
P = 35

**What if we run both maximum profit greedy and profit density greedy, then output the better result?**

**The approximation ratio is 2.**

**Proof:**

# The Knapsack Problem — 0-1 version

*NP-hard*

**Example :**

$n = 5,\ M = 11,$

$(p_1,\ p_2,\ p_3,\ p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2,\ 5,\ 6,\ 7)$

Solution is...? ( 0, 0, 1, 1, 0 )
P = 40

The greedy solution is
( 1, 1, 0, 0, 1 )
P = 35

**What if we run both maximum profit greedy and profit density greedy, then output the better result?**

**The approximation ratio is 2.**

**Proof:** $p_{max} \le P_{opt} \le P_{pd,\ frac}$

# The Knapsack Problem — 0-1 version

*NP-hard*

**Example :**

$n = 5,\ M = 11,$

$(p_1,\ p_2,\ p_3,\ p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2,\ 5,\ 6,\ 7)$

Solution is...? ( 0, 0, 1, 1, 0 )
P = 40

The greedy solution is
( 1, 1, 0, 0, 1 )
P = 35

**What if we run both maximum profit greedy and profit density greedy, then output the better result?**

**The approximation ratio is 2.**

**Proof:** $p_{max} \le P_{opt} \le P_{pd,\ frac}$

$p_{max} \le P_{mp}$

# The Knapsack Problem — 0-1 version

*NP-hard*

**Example :**

$n = 5,\ M = 11,$

$(p_1,\ p_2,\ p_3,\ p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2,\ \ 5,\ \ 6,\ \ 7)$

Solution is...? ( 0, 0, 1, 1, 0 )
P = 40

The greedy solution is
( 1, 1, 0, 0, 1 )
P = 35

**What if we run both maximum profit greedy and profit density greedy, then output the better result?**

**The approximation ratio is 2.**

**Proof:** $p_{max} \le P_{opt} \le P_{pd,\ frac}$

$p_{max} \le P_{mp}$

$P_{opt} \le P_{pd,\ frac} \le P_{pd,\ 0\text{-}1} + p_{max} \le P_{pd,\ 0\text{-}1} + P_{mp}$

# The Knapsack Problem — 0-1 version

*NP-hard*

**Example :**

$n = 5,\ M = 11,$

$(p_1,\ p_2,\ p_3,\ p_4, p_5) = (1, 6, 18, 22, 28)$

$(w_1, w_2, w_3, w_4, w_5) = (1, 2,\ \ 5,\ \ 6,\ \ 7)$

Solution is...? ( 0, 0, 1, 1, 0 )
P = 40

The greedy solution is
( 1, 1, 0, 0, 1 )
P = 35

**What if we run both maximum profit greedy and profit density greedy, then output the better result?**

**The approximation ratio is 2.**

**Proof:** $p_{max} \le P_{opt} \le P_{pd,\ frac}$

$p_{max} \le P_{mp}$

$P_{opt} \le P_{pd,\ frac} \le P_{pd,\ 0\text{-}1} + p_{max} \le P_{pd,\ 0\text{-}1} + P_{mp}$

Check https://www.cs.cornell.edu/courses/cs4820/2018fa/lectures/approx-onlinet.pdf

# Dynamic Programming

**Instance**: $n$ items $i$=1,2, ..., $n$;

weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;

knapsack capacity $B \in \mathbb{Z}^+$;

define:

$A(i, v)$ = minimum total weight of $S \subseteq \{1,2, \ldots, i\}$
with total value *exactly* $v$

$A(i, v) = \infty$ if no such $S$ exists

# Dynamic Programming

**Instance**: $n$ items $i$=1,2, ..., $n$;
weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;
knapsack capacity $B \in \mathbb{Z}^+$;

$A(i, v)$ = minimum total weight of $S \subseteq \{1,2, \ldots, i\}$
with total value *exactly* $v$

recursion:

$$A(i, v) = \min\{ A(i\text{-}1, v),\ A(i\text{-}1, v\text{-}v_i) + w_i \} \quad \text{for } i > 1$$

$$A(1, v) = \begin{cases} w_1 & \text{if } v = v_1 \\ \infty & \text{otherwise} \end{cases}$$

$1 \le i \le n, \quad 1 \le v \le V = \sum_i v_i$

Dynamic programming:
table size $O(nV)$
time complexity $O(nV)$

# Dynamic Programming

**Instance**: $n$ items $i$=1,2, ..., $n$;

weights $w_1, w_2, ..., w_n \in \mathbb{Z}^+$; values $v_1, v_2, ..., v_n \in \mathbb{Z}^+$;

knapsack capacity $B \in \mathbb{Z}^+$;

Find a subset of items whose total weight is bounded by $B$ and total value is maximized.

$A(i, v)$ = minimum total weight of $S \subseteq \{1,2, ..., i\}$ with total value *exactly* $v$

knapsack:

max $v$ that $A(n,v) \leq B$

Dynamic programming:

table size $O(nV)$

time complexity $O(nV)$

Polynomial Time?

# Polynomial Time

**Instance**: $n$ items $i$=1,2, ..., $n$;

weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;

knapsack capacity $B \in \mathbb{Z}^+$;

time complexity: $O(nV)$ where $V = \sum_i v_i$

- polynomial-time Algorithm $A$:

  ∃constant $c$, ∀ input $x \in \{0,1\}^*$, $A(x)$ terminates in $|x|^c$ steps

  $|x|$ = length of input $x$ (in *binary* code)

- pseudopolynomial-time Algorithm $A$:

  ∃constant $c$, ∀ input $x \in \{0,1\}^*$, $A(x)$ terminates in $|x|^c$ steps

  $|x|$ = length of input $x$ (in *unary* code)

# Dynamic Programming

**Instance**: $n$ items $i$=1,2, ..., $n$;

weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;

knapsack capacity $B \in \mathbb{Z}^+$;

Find a subset of items whose total weight is bounded by $B$ and total value is maximized.

$A(i, v)$ = minimum total weight of $S \subseteq \{1,2, \ldots, i\}$ with total value *exactly* $v$

$A(i, v) = \min\{ A(i\text{-}1, v),\ A(i\text{-}1, v\text{-}v_i) + w_i \}$

$$A(1, v) = \begin{cases} w_1 & \text{if } v = v_1 \\ \infty & \text{otherwise} \end{cases}$$

knapsack: $\max\{v: A(n,v) \le B\}$

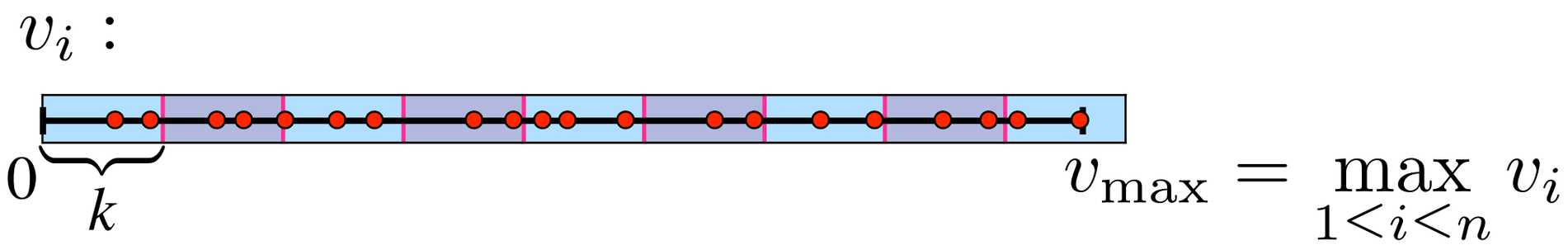
Dynamic programming time complexity $O(nV)$ where $V = \sum_i v_i$

**Pseudo-Polynomial Time!**

# Scaling & Rounding

**Instance**: $n$ items $i$=1,2, ..., $n$;
weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;
knapsack capacity $B \in \mathbb{Z}^+$;

Set $k$ = (to be fixed);
for $i$=1,2, ..., $n$, let $v'_i = \lfloor v_i / k \rfloor$;
return the knapsack solution found by dynamic programming with new values $v'_i$;



# Scaling & Rounding

**Instance**: $n$ items $i$=1,2, ..., $n$;
weights $w_1, w_2, ..., w_n \in \mathbb{Z}^+$; values $v_1, v_2, ..., v_n \in \mathbb{Z}^+$;
knapsack capacity $B \in \mathbb{Z}^+$;

Set $k$ = (to be fixed) ;
for $i$=1,2, ..., $n$, let $v'_i = \lfloor v_i / k \rfloor$;
return the knapsack solution found by dynamic programming with new values $v'_i$ ;

time complexity: $O(n\ V') = O(nV/k)$

where $V' = \sum_i v'_i = \sum_i \lfloor v_i / k \rfloor = O(V/k)$

and $V = \sum_i v_i$

**Instance**: $n$ items $i=1,2,\ldots,n$;

weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;

knapsack capacity $B \in \mathbb{Z}^+$;

Set $k$ = (to be fixed);

for $i=1,2,\ldots,n$, let $v'_i = \lfloor v_i / k \rfloor$;

return the knapsack solution found by dynamic programming with new values $v'_i$;

time complexity: $O(nV/k)$ where $V = \sum_i v_i$

$S^*$: optimal knapsack solution of the original instance

$$OPT = \sum_{i \in S^*} v_i = k \sum_{i \in S^*} \frac{v_i}{k} \leq k \sum_{i \in S^*} \left( \left\lfloor \frac{v_i}{k} \right\rfloor + 1 \right) \leq k \sum_{i \in S^*} \left\lfloor \frac{v_i}{k} \right\rfloor + nk$$

$S$: the solution returned by the algorithm

(optimal solution of the scaled instance) $\Rightarrow$ $\sum_{i \in S} \left\lfloor \frac{v_i}{k} \right\rfloor \geq \sum_{i \in S^*} \left\lfloor \frac{v_i}{k} \right\rfloor$

$$SOL = \sum_{i \in S} v_i \geq k \sum_{i \in S} \left\lfloor \frac{v_i}{k} \right\rfloor \geq k \sum_{i \in S^*} \left\lfloor \frac{v_i}{k} \right\rfloor \geq OPT - nk$$

**Instance**: $n$ items $i$=1,2, ..., $n$;

weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;

knapsack capacity $B \in \mathbb{Z}^+$;

Set $k$ = (to be fixed) ;

for $i$=1,2, ..., $n$, let $v'_i = \lfloor v_i / k \rfloor$;

return the knapsack solution found by dynamic programming with new values $v'_i$ ;

time complexity: $O(nV/k)$ where $V = \sum_i v_i \leq nv_{\max}$

$OPT$: optimal value of the original instance

$SOL$: value of the solution returned by the algorithm

$$SOL \geq OPT - nk \quad \Rightarrow \quad \frac{SOL}{OPT} \geq 1 - \frac{nk}{OPT} \geq 1 - \frac{nk}{v_{\max}}$$

WLOG: $OPT \geq v_{\max} = \max_{1 \leq i \leq n} v_i$

**Instance**: $n$ items $i$=1,2, ..., $n$;

weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;

knapsack capacity $B \in \mathbb{Z}^+$;

for any $0 \le \varepsilon \le 1$:

Set $k = \left\lfloor \frac{\epsilon v_{\max}}{n} \right\rfloor$ ; where $v_{\max} = \max_{1 \le i \le n} v_i$

for $i$=1,2, ..., $n$, let $v'_i = \lfloor v_i / k \rfloor$;

return the knapsack solution found by dynamic programming with new values $v'_i$ ;

time complexity: $O\left(\frac{n^2 v_{\max}}{k}\right) = O\left(\frac{n^3}{\epsilon}\right)$

*OPT*: optimal value of the original instance

*SOL*: value of the solution returned by the algorithm

$$\frac{SOL}{OPT} \ge 1 - \frac{nk}{v_{\max}} \ge 1 - \epsilon$$

# Approximation Ratio

Optimization problem:

- instance $I$:

  $\mathrm{OPT}(I)$ = optimum of instance $I$

- algorithm $A$: returns a solution $s$ for every instance $I$

  $\mathrm{SOL}_A(I)$ = value returned by $A$ on instance $I$

minimization: approximation ratio of algorithm $A$ is $\alpha$

if $\forall$ instance $I$ : $\dfrac{\mathrm{SOL}_A(I)}{\mathrm{OPT}(I)} \leq \alpha$

maximization: approximation ratio of algorithm $A$ is $\alpha$

if $\forall$ instance $I$ : $\dfrac{\mathrm{SOL}_A(I)}{\mathrm{OPT}(I)} \geq \alpha$

$\varepsilon$-approximation: $(1\text{-}\varepsilon)\ \mathrm{OPT}(I) \leq \mathrm{SOL}_A(I) \leq (1+\varepsilon)\ \mathrm{OPT}(I)$

(maximization) (minimization)

# Approximation Ratio

Optimization problem:

- instance $I$:

  $\mathrm{OPT}(I)$ = optimum of instance $I$

- algorithm $A$: returns a solution $s$ for every instance $I$ and $0 \le \varepsilon \le 1$

$\mathrm{SOL}_A(\varepsilon, I)$ = value returned by $A$ on instance $I$ and $\varepsilon$

- $A$ is a Polynomial-Time Approximation Scheme (PTAS) if:

  $\forall 0 \le \varepsilon \le 1$, $A$ returns in polynomial time and

$$(1-\varepsilon)\,\mathrm{OPT}(I) \le \mathrm{SOL}_A(\varepsilon, I) \le (1+\varepsilon)\,\mathrm{OPT}(I)$$

(maximization) (minimization)

- $A$ is a *Fully* Polynomial-Time Approximation Scheme (FPTAS) if:

  furthermore, $A$ returns in time $\mathrm{Poly}(1/\varepsilon, n)$ where $n = |I|$

(in *binary* code)

# Approximation Ratio

Optimization problem:

- instance $I$:

  $\mathrm{OPT}(I)$ = optimum of instance $I$

- algorithm $A$: returns a solution $s$ for every instance $I$ and $0 \le \varepsilon \le 1$

$\mathrm{SOL}_A(\varepsilon, I)$ = value returned by $A$ on instance $I$ and $\varepsilon$

- $A$ is a Polynomial-Time Approximation Scheme (PTAS) if:

  $\forall 0 \le \varepsilon \le 1$, $A$ returns in polynomial time and

$$(1\text{-}\varepsilon)\,\mathrm{OPT}(I) \le \mathrm{SOL}_A(\varepsilon, I) \le (1+\varepsilon)\,\mathrm{OPT}(I)$$

(maximization) (minimization)

- $A$ is a *Fully* Polynomial-Time Approximation Scheme (FPTAS) if:

  furthermore, $A$ returns in time $\mathrm{Poly}(1/\varepsilon, n)$ where $n = |I|$ (in *binary* code)

$O(n^{\frac{1}{\epsilon}})$ PTAS but not FPTAS $O(\frac{n}{\epsilon^2})$ PTAS and FPTAS

**Instance**: $n$ items $i$=1,2, ..., $n$;

weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;

knapsack capacity $B \in \mathbb{Z}^+$;

for any $0 \le \varepsilon \le 1$:

Set $k = \left\lfloor \frac{\epsilon v_{\max}}{n} \right\rfloor$ ; where $v_{\max} = \max_{1 \le i \le n} v_i$

for $i$=1,2, ..., $n$, let $v'_i = \lfloor v_i / k \rfloor$;

return the knapsack solution found by dynamic programming with new values $v'_i$ ;

time complexity: $O\left(\frac{n^3}{\epsilon}\right)$

approximation ratio: $\frac{SOL}{OPT} \ge 1 - \epsilon$

**FPTAS**

Are FPTASs the "best" approximation algorithms?

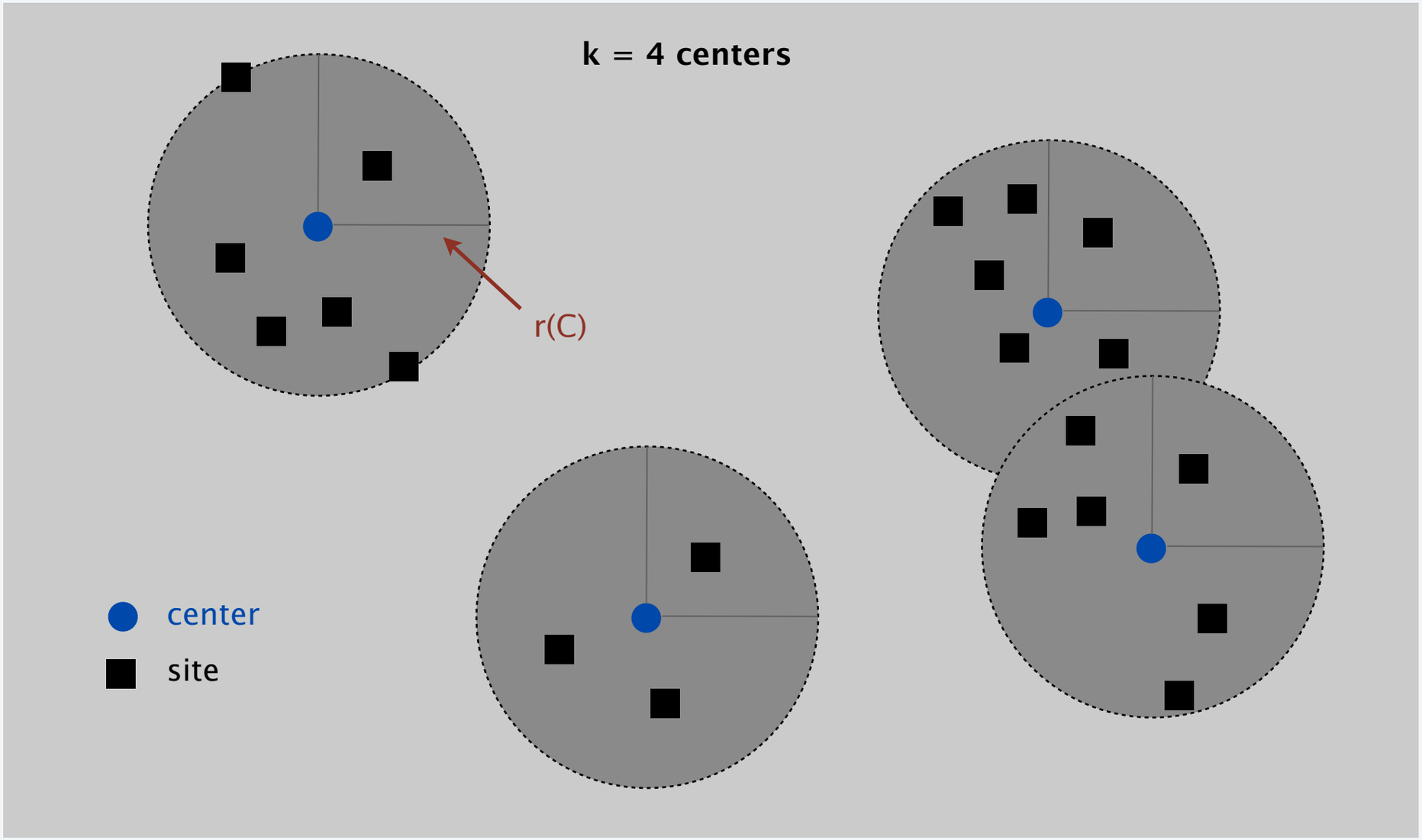
# Outline: Approximation Algorithms

- Bin packing
- 0-1 Knapsack
- K-center selection
- Take-home messages

# Center selection problem

Input. Set of $n$ sites $s_1, \ldots, s_n$ and an integer $k > 0$.

Center selection problem. Select set of $k$ centers $C$ so that maximum distance $r(C)$ from a site to nearest center is minimized.



# Center selection problem

Input. Set of $n$ sites $s_1, \ldots, s_n$ and an integer $k > 0$.

Center selection problem. Select set of $k$ centers $C$ so that maximum distance $r(C)$ from a site to nearest center is minimized.

# Center selection problem

Input. Set of $n$ sites $s_1, \ldots, s_n$ and an integer $k > 0$.

Center selection problem. Select set of $k$ centers $C$ so that maximum distance $r(C)$ from a site to nearest center is minimized.

Notation.

- $dist(x, y)$ = distance between sites $x$ and $y$.
- $dist(s_i, C) = \min_{c \in C} dist(s_i, c)$ = distance from $s_i$ to closest center.
- $r(C) = \max_i dist(s_i, C)$ = smallest covering radius.

Goal. Find set of centers $C$ that minimizes $r(C)$, subject to $|C| = k$.

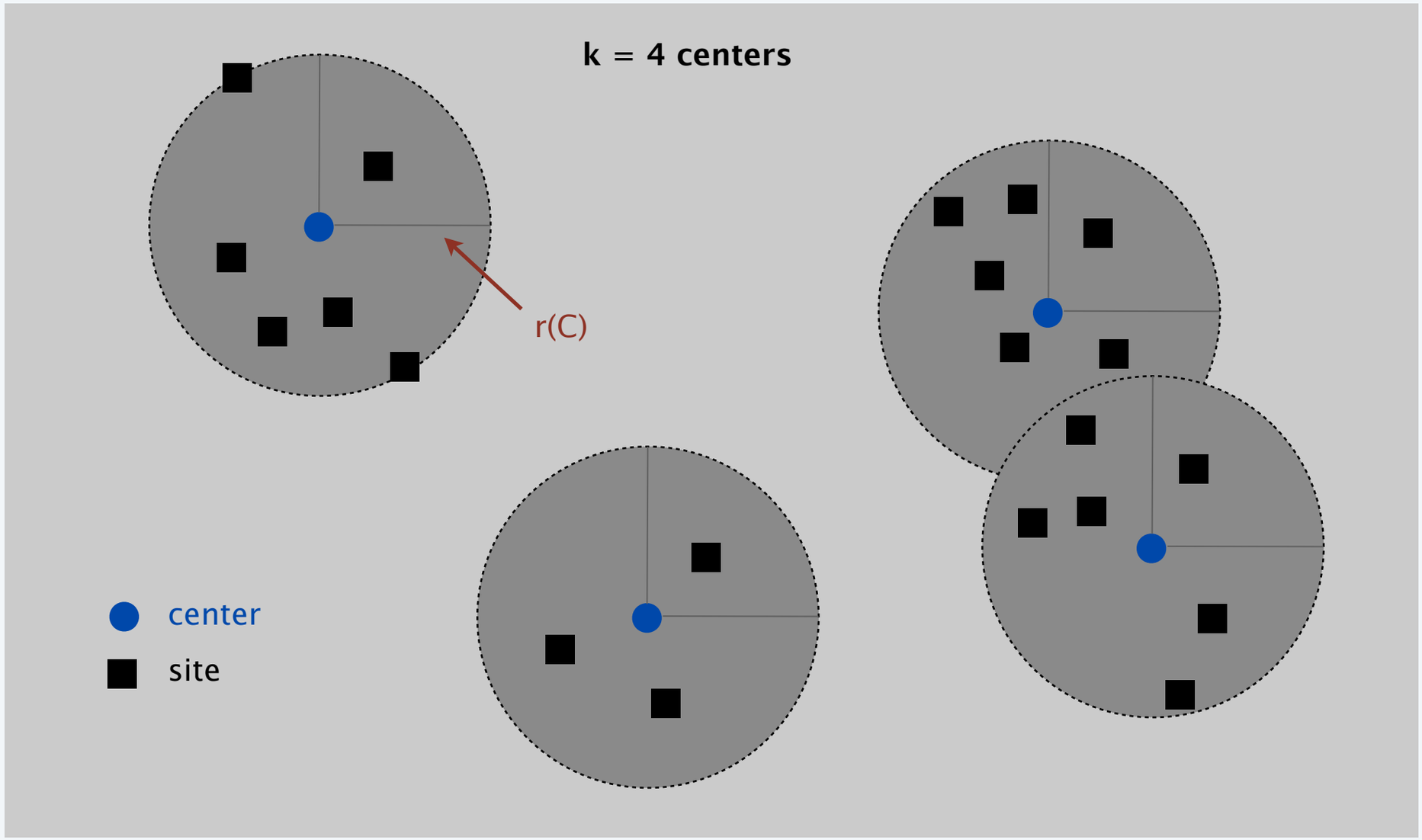
Distance function properties.

- $dist(x, x) = 0$ [ identity ]
- $dist(x, y) = dist(y, x)$ [ symmetry ]
- $dist(x, y) \leq dist(x, z) + dist(z, y)$ [ triangle inequality ]

# Center selection example

Ex: each site is a point in the plane, a center can be any point in the plane, $dist(x, y)$ = Euclidean distance.

Remark: search can be infinite!



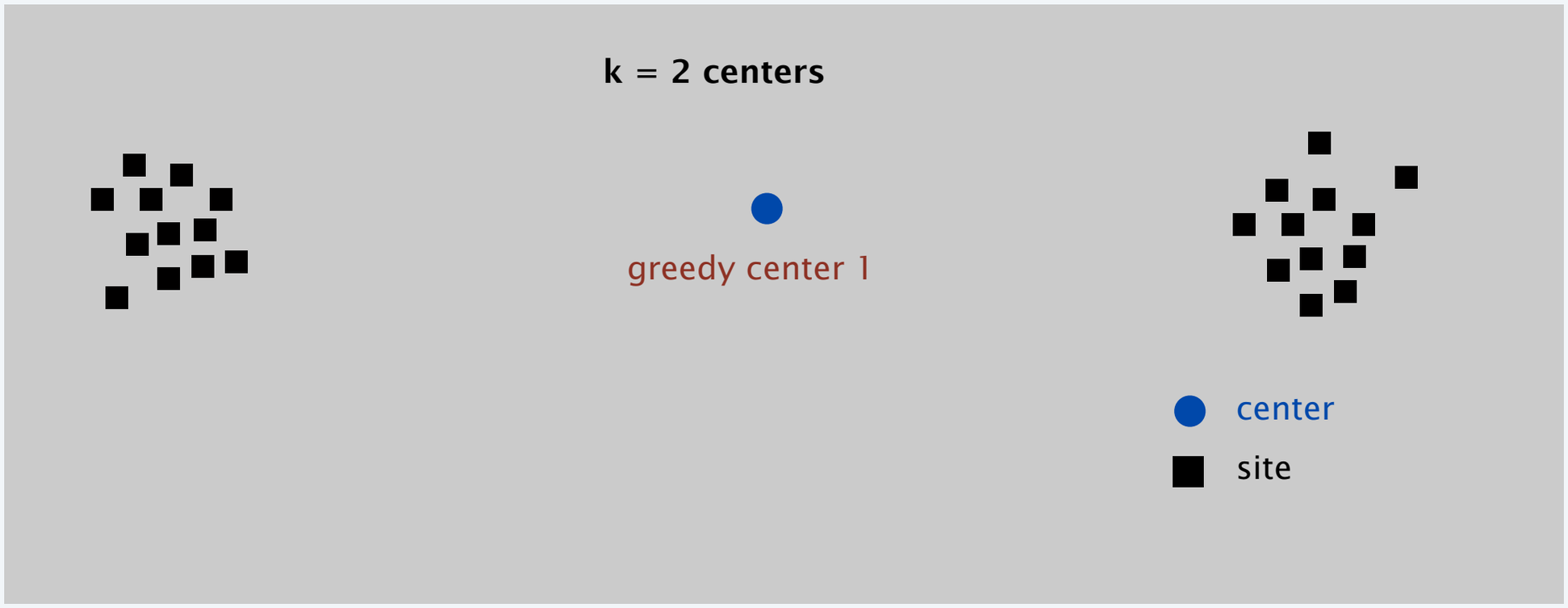
# Greedy algorithm: a false start

Greedy algorithm. Put the first center at the best possible location for a single center, and then keep adding centers so as to reduce the covering radius each time by as much as possible.

# Greedy algorithm: a false start

Greedy algorithm. Put the first center at the best possible location for a single center, and then keep adding centers so as to reduce the covering radius each time by as much as possible.

Remark: arbitrarily bad!



# ☞ A Greedy Solution — try again …

## ☞ A Greedy Solution — try again …

**What if we know that $r(C^*) \leq r$ where $C^*$ is the optimal solution set?**

## ☞ A Greedy Solution — try again …

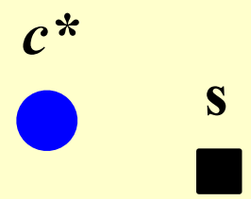
**What if we know that $r(C^*) \leq r$ where $C^*$ is the optimal solution set?**

s

■

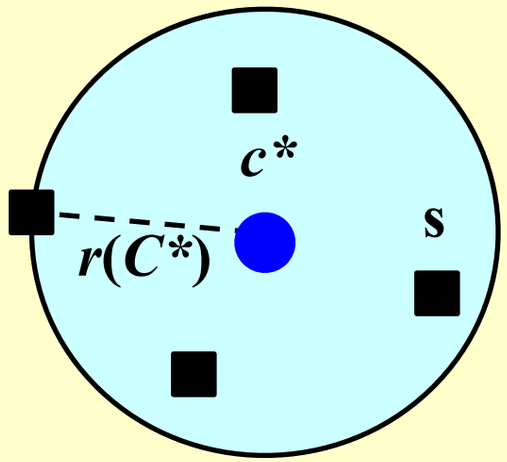
## ☞ A Greedy Solution — try again …

**What if we know that $r(C^*) \le r$ where $C^*$ is the optimal solution set?**



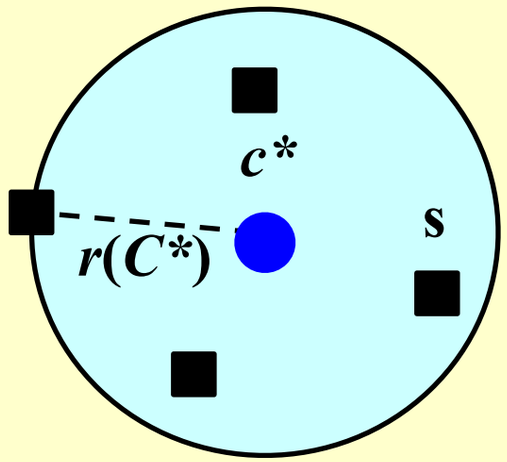
## ☞ A Greedy Solution — try again …

**What if we know that $r(C^*) \le r$ where $C^*$ is the optimal solution set?**



## ☞ A Greedy Solution — try again …

**What if we know that $r(C^*) \le r$ where $C^*$ is the optimal solution set?**

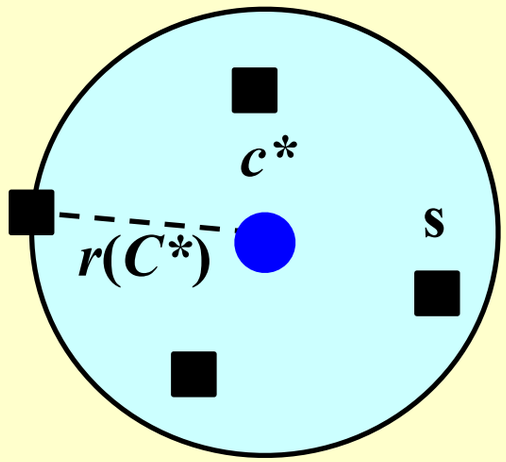



**Discussion 15:**

Take s to be the center, how can we select $r$ so that s can cover all the sites that are covered by $c^*$?

## ☞ A Greedy Solution — try again …

**What if we know that $r(C^*) \le r$ where $C^*$ is the optimal solution set?**



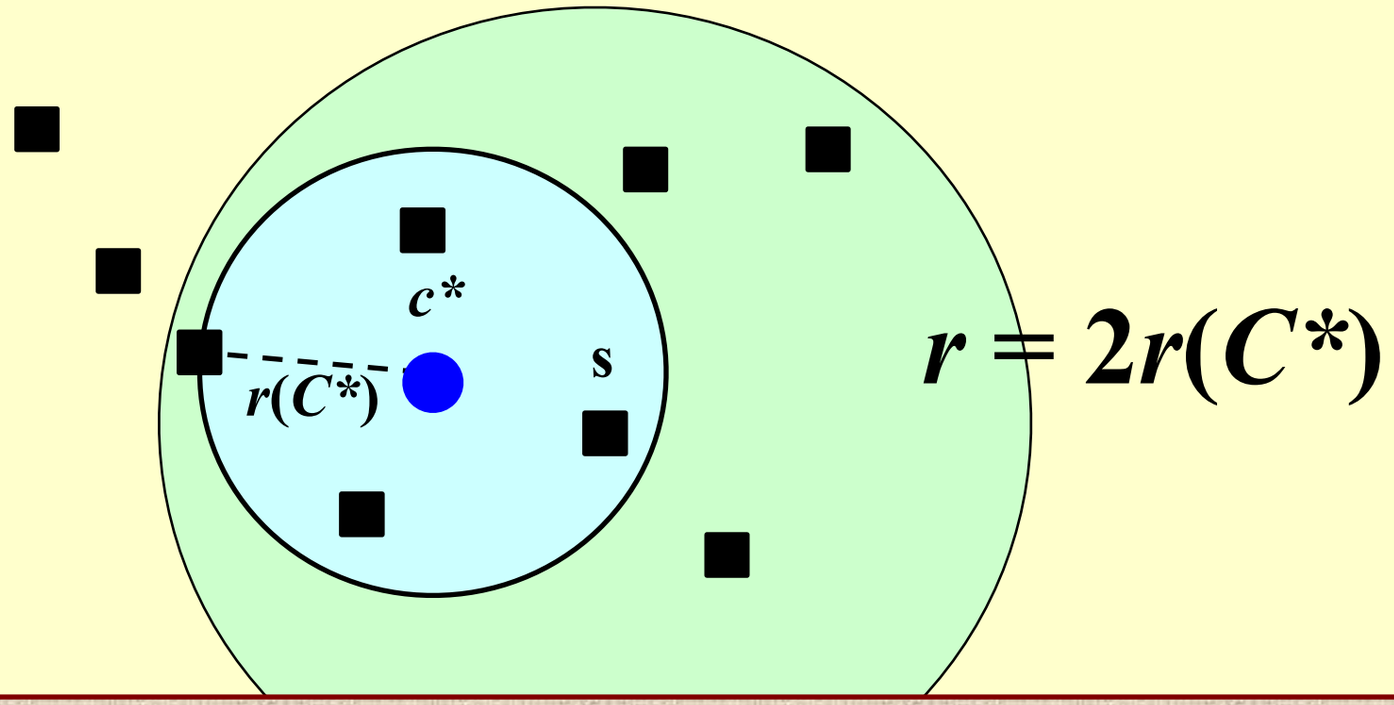

$$r = 2r(C^*)$$

**Discussion 15:**

Take s to be the center, how can we select $r$ so that s can cover all the sites that are covered by $c^*$?

## ☞ A Greedy Solution — try again …

**What if we know that $r(C^*) \le r$ where $C^*$ is the optimal solution set?**




**Discussion 15:**

Take s to be the center, how can we select $r$ so that s can cover all the sites that are covered by $c^*$?

```
Centers  Greedy-2r ( Sites S[ ], int n, int K, double r*)
{   Sites  S'[ ] = S[ ]; /* S' is the set of the remaining sites */
    Centers  C[ ] = ∅;
    while ( S'[ ] != ∅ ) {
        Select any s from S' and add it to C;
        Delete all s' from S' that are at dist(s', s) ≤ 2r*;
    } /* end-while */
    if ( |C| ≤ K ) return C;
    else ERROR(No set of K centers with covering radius at most r*);
}
```

```
Centers  Greedy-2r ( Sites S[ ], int n, int K, double r*)
{   Sites  S'[ ] = S[ ]; /* S' is the set of the remaining sites */
    Centers  C[ ] = ∅;
    while ( S'[ ] != ∅ ) {
        Select any s from S' and add it to C;
        Delete all s' from S' that are at dist(s', s) ≤ 2r*;
    } /* end-while */
    if ( |C| ≤ K ) return C;
    else ERROR(No set of K centers with covering radius at most r*);
}
```

**【Theorem】 Suppose the algorithm selects more than *K* centers. Then for any set *C* of size at most *K*, the covering radius is $r(C) > r^*$.**

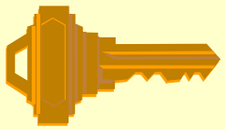
```
Centers  Greedy-2r ( Sites S[ ], int n, int K, double r*)
{   Sites  S'[ ] = S[ ]; /* S' is the set of the remaining sites */
    Centers  C[ ] = ∅;
    while ( S'[ ] != ∅ ) {
        Select any s from S' and add it to C;
        Delete all s' from S' that are at dist(s', s) ≤ 2r*;
    } /* end-while */
    if ( |C| ≤ K ) return C;
    else ERROR(No set of K centers with covering radius at most r*);
}
```

**【Theorem】 Suppose the algorithm selects more than *K* centers. Then for any set *C* of size at most *K*, the covering radius is $r(C) > r^*$.**

Proof see Algorithm Design Sec 11.2

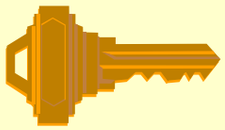
# Do we really know $r(C^*)$ ?

# Do we really know $r(C^*)$ ?



**Binary search for $r$**

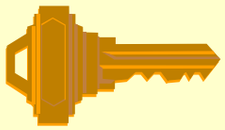
# Do we really know $r(C^*)$ ?



**Binary search for $r$**

$$0 < r \leq r_{max}$$

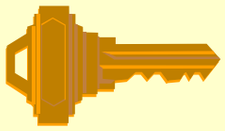
# Do we really know $r(C^*)$ ?



**Binary search for $r$**

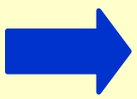
$0 < r \leq r_{max}$ **Guess:** $r = ( 0 + r_{max} ) / 2$

# Do we really know $r(C^*)$ ?



**Binary search for $r$**

$0 < r \leq r_{max}$ **Guess:** $r = ( 0 + r_{max} ) / 2$

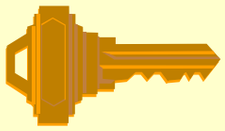


**Yes: $K$ centers found with $2r$**

**or**

**No: $r$ is too small**

# Do we really know $r(C^*)$ ?



**Binary search for $r$**

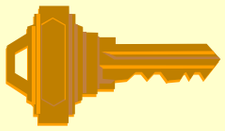
$0 < r \leq r_{max}$ **Guess:** $r = ( 0 + r_{max} ) / 2$

**Yes: $K$ centers found with $2r$**

**or**

**No: $r$ is too small**

# Do we really know $r(C^*)$ ?



**Binary search for $r$**

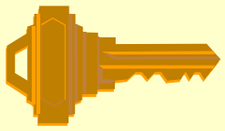
$0 < r \leq r_{max}$ **Guess:** $r = ( 0 + r_{max} ) / 2$

**Yes: $K$ centers found with $2r$**

**or**

**No: $r$ is too small**

# Do we really know $r(C^*)$ ?



**Binary search for $r$**

$0 < r \leq r_{max}$ **Guess:** $r = ( 0 + r_{max} ) / 2$

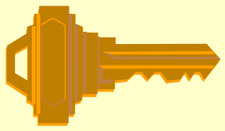
**Yes: $K$ centers found with $2r$**

**or**

**No: $r$ is too small**

$r_0 < r \leq r_1$

# Do we really know $r(C^*)$ ?



**Binary search for $r$**

$0 < r \le r_{max}$ **Guess:** $r = ( 0 + r_{max} ) / 2$

**Yes: $K$ centers found with $2r$**

**or**

**No: $r$ is too small**

$r_0 < r \le r_1$ $r = ( r_0 + r_1 ) / 2$

# Do we really know $r(C^*)$ ?

Binary search for $r$

$0 < r \le r_{max}$ **Guess:** $r = ( 0 + r_{max} ) / 2$

**Yes: $K$ centers found with $2r$**

**or**

**No: $r$ is too small**

$r_0 < r \le r_1$ $r = ( r_0 + r_1 ) / 2$

**Solution radius = $2r_1$ —— close to the true 2-approximation solution**

# Do we really know $r(C^*)$ ?

Binary search for $r$

$0 < r \le r_{max}$ Guess: $r = ( 0 + r_{max} ) / 2$

Yes: $K$ centers found with $2r$

or

No: $r$ is too small

$r_0 < r \le r_1$ $r = ( r_0 + r_1 ) / 2$

Solution radius = $2r_1$ —— close to the true 2-approximation solution

We have a smarter solution without using the input r.

# Center selection: greedy algorithm

Repeatedly choose next center to be site farthest from any existing center.

*GREEDY-CENTER-SELECTION* $(k, n, s_1, s_2, \ldots, s_n)$

$C \leftarrow \emptyset$.

*REPEAT* $k$ *times*

*Select a site* $s_i$ *with maximum distance* $dist(s_i, C)$.

$C \leftarrow C \cup s_i$.

*RETURN* $C$.

site farthest from any center

**Property.** Upon termination, all centers in $C$ are pairwise at least $r(C)$ apart.

**Pf.** By construction of algorithm.

# Center selection: analysis of greedy algorithm

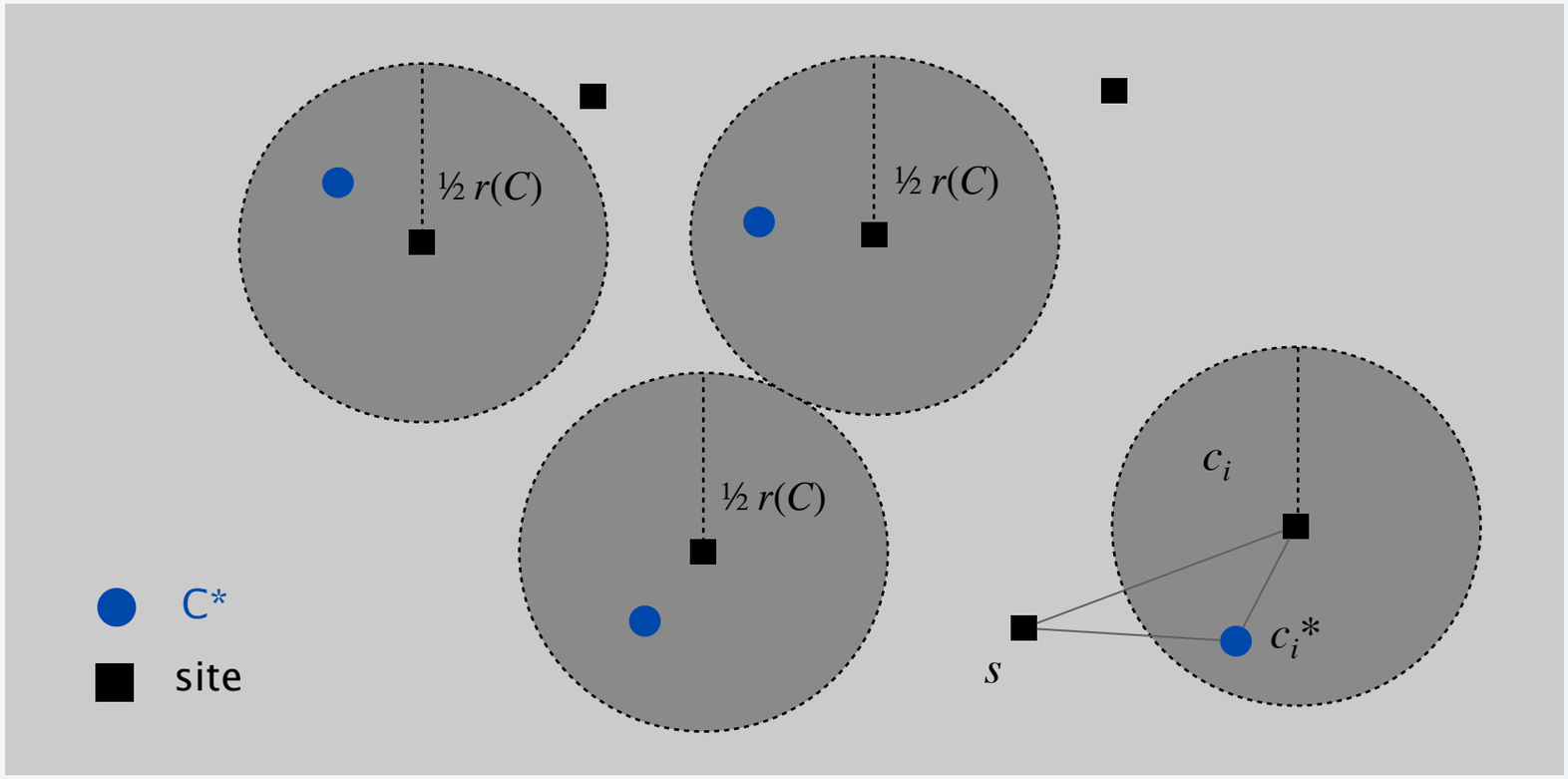
Lemma. Let $C^*$ be an optimal set of centers. Then $r(C) \leq 2r(C^*)$.

Pf. [by contradiction] Assume $r(C^*) < \frac{1}{2}\, r(C)$.

- For each site $c_i \in C$, consider ball of radius $\frac{1}{2}\, r(C)$ around it.
- Exactly one $c_i^*$ in each ball; let $c_i$ be the site paired with $c_i^*$.
- Consider any site $s$ and its closest center $c_i^* \in C^*$.
- $dist(s, C) \leq dist(s, c_i) \leq dist(s, c_i^*) + dist(c_i^*, c_i) \leq 2r(C^*)$.
- Thus, $r(C) \leq 2r(C^*)$. ▪

$\leq$ r(C*) since $c_i^*$ is closest center

Δ-inequality



# Center selection

Lemma. Let $C^*$ be an optimal set of centers. Then $r(C) \leq 2r(C^*)$.

Theorem. Greedy algorithm is a 2-approximation for center selection problem.

Remark. Greedy algorithm always places centers at sites, but is still within a factor of 2 of best solution that is allowed to place centers anywhere.

e.g., points in the plane

Question. Is there hope of a 3/2-approximation? 4/3?

# Dominating set reduces to center selection

Theorem. Unless **P** = **NP**, there no ρ-approximation for center selection problem for any ρ < 2.

Pf. We show how we could use a $(2-\varepsilon)$ approximation algorithm for CENTER-SELECTION selection to solve DOMINATING-SET in poly-time.

- Let $G = (V, E)$, $k$ be an instance of DOMINATING-SET.
- Construct instance $G'$ of CENTER-SELECTION with sites $V$ and distances
  - $dist(u, v) = 1$ if $(u, v) \in E$
  - $dist(u, v) = 2$ if $(u, v) \notin E$
- Note that $G'$ satisfies the triangle inequality.
- $G$ has dominating set of size $k$ iff there exists $k$ centers $C^*$ with $r(C^*) = 1$.
- Thus, if $G$ has a dominating set of size $k$, a $(2-\varepsilon)$-approximation algorithm for CENTER-SELECTION would find a solution $C^*$ with $r(C^*) = 1$ since it cannot use any edge of distance $2$. ▪

# Outline: Approximation Algorithms

- Bin packing
- 0-1 Knapsack
- K-center selection
- Take-home messages

# Take-Home Messages

- Approximation algorithms: Deal with computational intractable (NP-hard) optimization problems, by relaxation on the optimality of solution to be find.

- Some definitions: approximation ratio, PTAS, and FPTAS.

- Two techniques in this lecture: Greedy search and rounding dynamic programming.

- Interesting in math properties: comparison on the unknown optimal solution, proving lower bounds.

# Research Project 6
## Texture Packing (26)

# Research Project 6
## Texture Packing (26)

**Texture Packing is to pack multiple rectangle shaped textures into one large texture. The resulting texture must have a given width and a minimum height.**

**You are to design and analyze an approximation algorithm that runs in polynomial time.**

**Detailed requirements can be downloaded from**
**https://pintia.cn/**

Thanks for your attention!
Discussions?

# Reference

Data Structure and Algorithm Analysis in C (2nd Edition)： Sec 10.1.3.

Algorithm design: Chap. 11.

Introduction to Algorithms (4th Edition): Chap. 35.

Slides from Kevin Wayne: https://www.cs.princeton.edu/~wayne/kleinberg-tardos/

Yitong Yin's lecture:

https://tcs.nju.edu.cn/wiki/index.php?title=%E9%AB%98%E7%BA%A7%E7%AE%97%E6%B3%95_(Fall_2020)

https://tcs.nju.edu.cn/slides/aa2020/Greedy.pdf

https://tcs.nju.edu.cn/slides/aa2020/DP.pdf

Approximation Algorithms book: https://ics.uci.edu/~vazirani/book.pdf Chap. 8, 9.